Introduction au calcul parallèle

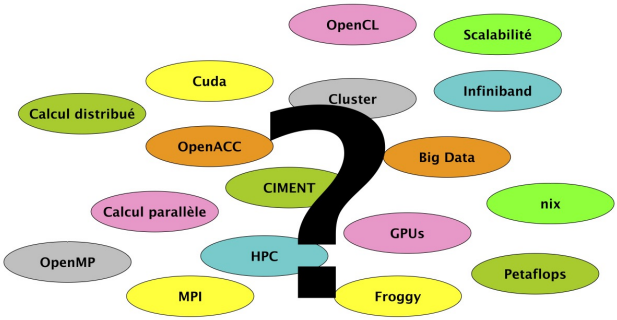
Céline Acary-Robert, Frédéric Audra, Pierre-Antoine Bouttier, Glenn Cougoulat, Albanne Lecointre, Violaine Louvet, Franck Pérignon

février-mars 2023

Généralités autour du calcul parallèle

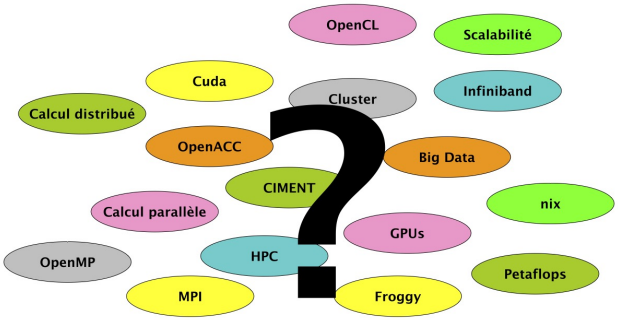
Violaine Louvet, Franck Pérignon

février-mars 2023



Calcul parallèle ?
Pour quoi faire ?
Comment ?
Quels modèles, quels langages ?
...

Introduction
Les bases, le vocabulaire, les concepts, les pistes pour aller plus loin, une approche pratique d'OpenMP et MPI.



Calcul parallèle ?
Pour quoi faire ?
Comment ?
Quels modèles, quels langages ?
...

Introduction
Les bases, le vocabulaire, les concepts, les pistes pour aller plus loin, une approche pratique d'OpenMP et MPI.

Notre but (souhait ...)
⇒ Concrétiser pour vous ce qu'est le "calcul parallèle", vous donner les bases et les pistes pour approfondir et appliquer cela à vos codes.
⇒ Vous apprendre à utiliser les ressources en calcul à Grenoble.

Contexte, pourquoi ce cours?

Ce module **Introduction au calcul parallèle** fait partie d'un ensemble de formations transverses proposées par le collège doctoral, en partenariat avec GRICAD et MaiMoSiNE, intitulé **Outils pour le traitement de données, le développement logiciel et le calcul scientifique**.
Pour commencer, nous allons donc :

- présenter le contexte de ces cours,
- présenter brièvement GRICAD et MaiMoSiNE.

Formations transverses CED - Outils pour le traitement de données, le développement logiciel et le calcul scientifique

Accès aux documents (ce cours et les autres):

https://pole-calcul-formation.gricad-pages.univ-grenoble-alpes.fr/ced/

Les formations :

- Les bases du systeme linux pour le calcul scientifique
- Gestion de projets et developpement collaboratifs - utilisation de la plateforme gricad-gitlab
- Des sources a l'executable : la chaine de compilation
- Introduction au calcul parallele
- Introduction aux formats de donnees et a la visualisation en calcul scientifique - entrees-sorties paralleles

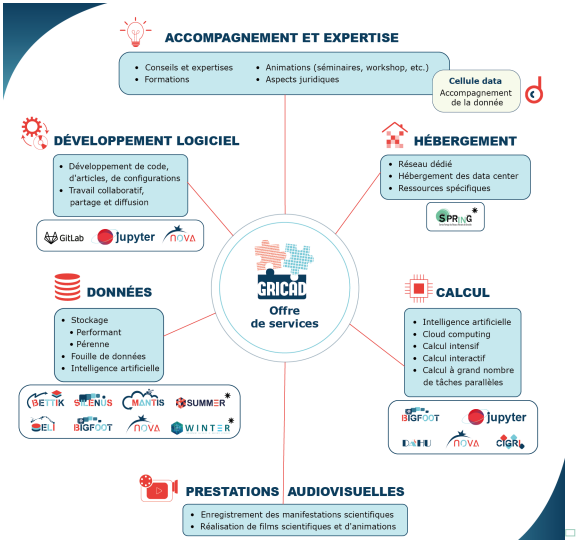
L'UAR GRICAD

Unité multi-tutelles (CNRS, UGA, GINP, INRIA) d'Appui et de Recherche
GRenoble Infra CAlcul et Données
Missions:

- **Accompagnement et conseils** aux chercheurs sur leurs besoins liés **au calcul et aux données**
- **Mise à disposition à l'ensemble des chercheurs et personnels en support de la recherche** d'infrastructures avancées et mutualisées pour le calcul intensif et l'exploitation des données de la recherche
- Participation aux infrastructures de site en terme d'hébergement, de stockage et de virtualisation

- Interaction forte avec les laboratoires : postes mutualisés, lettres de mission
- 2ème étage du batiment IMAG.
- *https://gricad.univ-grenoble-alpes.fr/*
- **Contact :** gricad-contact@univ-grenoble-alpes.fr

Les services proposés par GriCAD



✱ Services DGDSI/UGA

Maison de la Modélisation et de la Simulation Nanosciences et Environnement

MaiMoSINE est une Structure Fédérative de Recherche de l'Université Grenoble Alpes. Sa mission principale est de favoriser les interactions entre le monde socio-économique et les chercheurs de l'Université Grenoble Alpes dans le domaine des mathématiques et de l'informatique, en particulier en science des données, modélisation, simulation et calcul intensif.

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

5 Comment paralléliser ?

6 Compléments

7 Conclusion

Introduction au calcul parallèle - Contexte et motivations

Point de départ : une application, un code de calcul, *fonctionnel et optimisé*, qui tourne en *séquentiel*.

Calcul séquentiel *: exécution d'instructions étape par étape, que les opérations soient indépendantes ou non, a priori sur une seule ressource.*

Introduction au calcul parallèle - Contexte et motivations

Point de départ : une application, un code de calcul, *fonctionnel et optimisé*, qui tourne en *séquentiel*.

Calcul séquentiel *: exécution d'instructions étape par étape, que les opérations soient indépendantes ou non, a priori sur une seule ressource.*

Mais ...

- Ce code a atteint ses limites de performances
  - il est trop coûteux en temps de calcul, en mémoire,
  - les volumes de données à traiter sont trop importants, trop longs à écrire,
  - les performances sont moins bonnes que sur des machines plus vieilles, ...
- Volonté ou nécessité d'exploiter de nouvelles ressources
  - mutualisation de ressources (cluster ...),
  - problématique du coût financier et/ou environnemental (conso électrique etc),

Bref votre code n'est plus adapté aux nouvelles ressources et/ou ses performances ne sont pas satisfaisantes. Quelles solutions ? $\Rightarrow$ *HPC et calcul parallèle*.

HPC : High Performance Computing

- Exploiter au mieux les ressources, améliorer les performances : vitesse (Flops ...), mémoire, réseau, disque.
- Paralléliser

Calculer en parallèle : *effectuer plusieurs calculs simultanés sur différentes ressources.*

Comment ? Par *décomposition* du problème en plusieurs sous problèmes plus "petits".

- **data parallelism** ("fine grain") : découper/ distribuer les données ,
- **task parallelism** ("coarse grain") : identifier des tâches indépendantes ,

$\Rightarrow$ plusieurs tâches ou sous-problèmes plus ou moins autonomes qui pourront être traités simultanément, de manière asynchrone.

Mots-clés : distribution des données, distribution des tâches, communication entre les ressources (échanges), synchronisation ...

Quels bénéfices attendre de la parallélisation ?

Améliorer les performances

- Aller plus vite ! Obtenir une solution au problème plus rapidement.
  dans l'idéal : $temps_sequentiel \rightarrow temps_parallele = \frac{temps_sequentiel}{nombre_ressources}$
- Résoudre des problèmes (beaucoup) plus grands.
- Traiter des volumes de données plus importants.

Mieux (?) exploiter les ressources

- Mutualiser, partager les ressources.
- Baisser les coûts, la consommation électrique ?
- ...

Mais ce n'est pas gratuit !

Pour quel prix ?

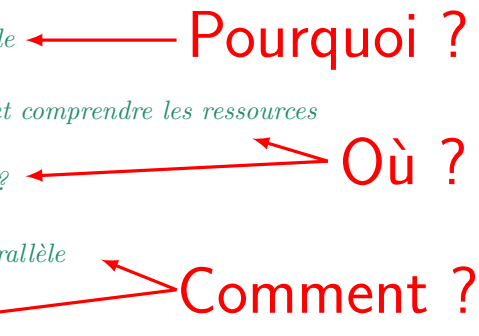
- Un prix à payer en terme de développements.
  - Conception du logiciel à revoir (probablement).
  - Adaptation nécessaire des algorithmes et des méthodes utilisées.
  - Apprentissage de nouveaux langages, outils (debug ...).
- Utilisation/exploitation plus complexe.
- Très dépendant des architectures : il faudra les connaître, avoir une idée de leur évolution etc. La portabilité et la modularité deviennent cruciales.

En conséquence, **avant de pouvoir paralléliser** il sera nécessaire de

- Comprendre et connaître l'architecture des ressources disponibles, pour les choisir correctement et les exploiter pleinement.
- Comprendre le comportement du programme, des algorithmes, pour pouvoir l'adapter, choisir les bonnes méthodes (potentiellement différentes de celles utilisées en séquentiel).
- Connaître les modèles de programmation parallèle.

Organisation de la séance

1. Contexte et présentation du module
2. Avant de paralléliser : connaitre et comprendre les ressources
3. Quelles architectures aujourd'hui ?
4. Les modèles de programmation parallèle
5. Comment paralléliser ?
6. Compléments
7. Conclusion



Organisation du module et planning

1. Généralités autour du calcul parallèle (Séance 1 - 01/03/2023)
2. Accès et utilisation des ressources grenobloises (Séance 2 - 02/03/2023)
3. Calcul en mémoire partagée
   - Généralités (séance 3)
   - OpenMP (Séances 3 et 4 - 03,10/03/2023)
   - Calcul sur carte graphique (Séances 5 et 6 - 14,16/03/2023)
4. Calcul en mémoire distribuée
   - MPI (Séances 4 à 8 - 22,24,29,30,31/03/2023)
   - Calcul sur grille (Séance 9 - 06/04/2023)

Planning : voir *https://pole-calcul-formation.gricad-pages.univ-grenoble-alpes.fr/ced/infos_pratiques/*

Infos pratiques

- **Salle, accès et contacts** : voir
  https://pole-calcul-formation.gricad-pages.univ-grenoble-alpes.fr/ced/infos_pratiques/.
- **Accès aux documents** - Tous les cours sont accessibles sur le site :
  https://pole-calcul-formation.gricad-pages.univ-grenoble-alpes.fr/ced/
  Remarque: ces documents seront mis à jour au fur et à mesure des séances. Pensez à toujours récupérer la dernière version.
  Pour accéder aux exercices et solutions, il sera nécessaire de posséder un compte sur la plateforme *https://gricad-gitlab.univ-grenoble-alpes.fr* et de faire partie du projet trainings du groupe pole-calcul-formation/introduction-au-calcul-parallele.

Rappel des pré-requis indispensables

- Etre à l'aise avec un système unix/linux (WSL pour Windows ?) et la ligne de commande.
- Savoir utiliser la plateforme gricad-gitlab et git (indispensable pour les exercices et démos)
- Maitriser la chaîne de production d'un exécutable (compilation/édition de lien) et connaitre les bases de l'utilisation de CMake
- Maîtriser au moins un langage de programmation (C/C++ ou Fortran). Les exercices seront dans l'un ou l'autre langage, parfois les deux.
  http://www.idris.fr/formations/fortran/
  http://www.idris.fr/formations/c/
- Et surtout, connaître son identifiant agalan et le password associé !

Pour les trois premiers points, voir les différents modules disponibles dans *https://pole-calcul-formation.gricad-pages.univ-grenoble-alpes.fr/ced/plans_modules/*.

Tour de table

Merci de vous présenter brièvement (école doctorale, sujet de thèse ...)

- Quelle est votre expérience en calcul parallèle ?
- Qu'attendez vous de ce module (Culture générale, un cas d'usage concret ...) ?
- Avez-vous suivi le module Linux, le module gitlab/git ?

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

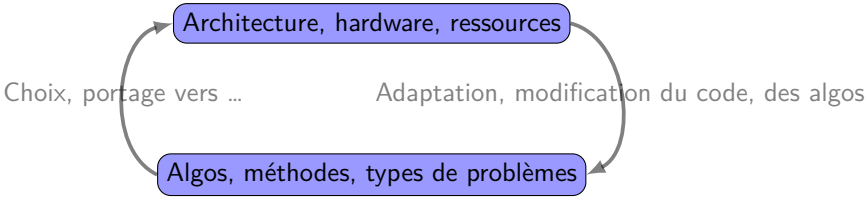
5 Comment paralléliser ?

6 Compléments

7 Conclusion

Comment paralléliser ?

Première étape indispensable : *connaître et comprendre les ressources, les architectures*.



Nous allons passer en revue les différentes “familles” de systèmes/architectures et introduire tout le vocabulaire nécessaire à une bonne compréhension du parallélisme.

1. Les **ressources** : lexique, fonctionnement d'un processeur, principales notions “hardware” à connaître.
2. **Architectures disponibles** : état des lieux, évolutions et tendances.
3. Quels **types de parallélisme** pour quelles architectures ?

Les grands types d'usage du calcul

High Performance Computing / HPC

- Simulations numériques massives
- Physique, climat, chimie ...
- Machine souvent spécifique avec un réseau ultra-performant

Data Analytics / Big Data

- Analyse de grands ensembles de données de type variés
- SHS, santé, ...
- Serveurs classiques distribués avec pas mal de mémoire, avec une pile logicielle dédiée utilisant des bases de données spécifiques (Hadoop, nosql, graph ...)
- ML/DL : GPU

High Throughtput Computing / Grille

- Traitement de données structurées distribuées
- Physique des Hautes Energies, Bioinformatique, ...
- Serveurs classiques distribués avec un middlware spécifique

Cloud Computing

- Ressources informatiques à la demande
- Usages très variés
- Serveurs classiques boostés au niveau CPU et mémoire, avec une technologie de virtualisation

Les grands types d'usage du calcul

High Performance Computing / HPC

- Simulations numériques massives
- Physique, climat, chimie ...
- Machine souvent spécifique avec un réseau ultra-performant

Data Analytics / Big Data

- Analyse de grands ensembles de données de type variés
- SHS, santé, ...
- Serveurs classiques distribués avec pas mal de mémoire, avec une pile logicielle dédiée utilisant des bases de données spécifiques (Hadoop, nosql, graph ...)
- **ML/DL : GPU**

High Throughtput Computing / Grille

- Traitement de données structurées distribuées
- Physique des Hautes Energies, Bioinformatique, ...
- Serveurs classiques distribués avec un middlware spécifique

Cloud Computing

- Ressources informatiques à la demande
- Usages très variés
- Serveurs classiques boostés au niveau CPU et mémoire, avec une technologie de virtualisation

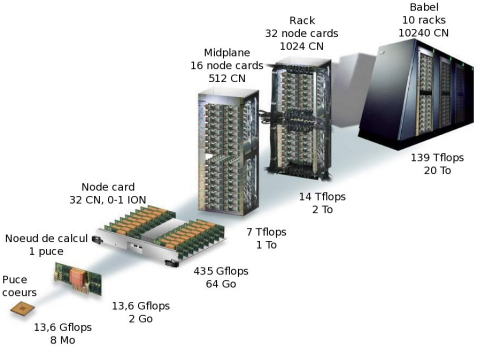
Quels parallélismes pour quelles architectures ?

Nous allons voir qu'il existe différents types de parallélismes, fortement lié à l'architecture du "calculateur" utilisé.

- Parallélisme 'hardware', au niveau du processeur (instructions ...).
- Parallélisme par échange de messages, entre des "noeuds" de calcul, *systèmes à mémoire distribuée*.
- Parallélisme intra-noeud, *systèmes à mémoire partagée*.
- Utilisation d'*accélérateurs* (GPUs).
- Parallélisme par vectorisation, "SIMD".
- Parallélisme "embarassant" (embarassingly parallel).

Nous allons aborder ces différents points, d'abord du point de vue "hardware" puis ensuite du point de vue des modèles et langages.

Comprendre et connaître l'architecture des ressources



Avant de paralléliser, un point sur ce qu'il faut connaître: vocabulaire, système, hardware, ...

- une partie qui **calcule** : horloge, ALU, FPU, cycle, pile d'instruction ...
- une partie qui **lit/écrit** : mémoire, hiérarchie, cache, registre
- une partie qui **discute, échange** : bus, réseau
- une partie qui **sauvegarde** : disque, système de fichiers

Comprendre et connaitre l'architecture des ressources

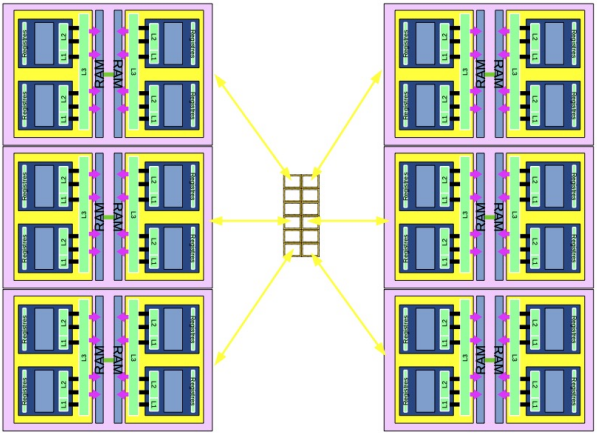
- **bit (b)** : binary digit, plus petite unité d'information d'un composant en informatique. Vaut 0 ou 1
  $\longrightarrow$ 1 bit permet donc d'avoir 2 états (0 ou 1), 2 bits permettent d'avoir 4 (2^2) états (00,01,10,11),... , n bits permettent ainsi d'avoir 2^n états
- **octet (o) ou byte (B)** = composé de 8 bits

Usage

Processeurs et mémoires opèrent sur des **octets**
Débits des réseaux et des bus exprimés en **bits par seconde**

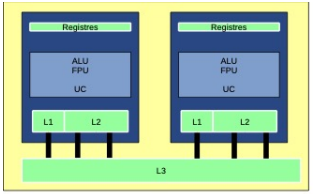
L'objectif

Connaître les éléments d'un supercalculateur



Architecture d'un **processeur (CPU)** [1]

Processeur bi-cœur



Le *processeur/CPU* est l'ensemble matériel qui fournit la puissance de calcul.
Sa puissance dépend:

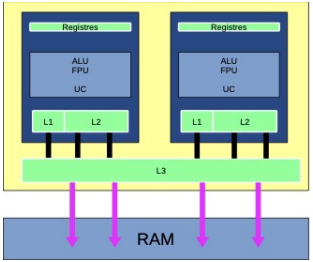
- du nombre de *cœurs*: noyaux de calcul,
- de la taille de sa *mémoire cache* (différents niveaux: L1, L2, L3),
- de sa *fréquence d'horloge* (mesurée en GHz).

On trouve aussi:

- l'*UC* (Unité de Contrôle) qui permet de synchroniser les différents éléments du processeur,
- l'*ALU* (Unité d'Arithmétique et Logique) prend en charge les calculs arithmétiques,
- la *FPU* (Unité de Calcul Flottant) prend en charge les calculs flottants,
- les *registres*, qui sont des mémoires de petite taille (quelques octets), qui permettent de garder les informations proche de l'*ALU*.

[1] *Central Processing Unit*

Fonctionnement d'un processeur - Fréquence et cycle horloge



- Pour calculer, le **CPU** échange des données et des instructions entre la **mémoire principale (RAM)** et les unités qui calculent (ALU, FPU).
  Son activité est cadencée par une horloge interne.
- **Jeu d'instructions** : ensemble des opérations, plus ou moins complexes, qu'un processeur peut exécuter.
  Types d'instruction:
  - Accès mémoire/transfert
  - Opérations arithmétiques (`+`, `-`, `x`, …)
  - Opérations logiques (`AND`, `OR`, …)
  - Contrôle (`if`, …)

Puissance d'un processeur - Métriques

Fréquence d'horloge (cycle)

Nombre d'impulsions envoyées par seconde (Hz).
1 impulsion = 1 action (instruction ou partie d'instruction).
Exemple: Un processeur d'une fréquence de 3GHz peut réaliser $3x10^9$ impulsions/seconde.
On parle souvent de Million d'Instructions Par Seconde : **MIPS**

Cycles Par Instruction(CPI)

Le nombre de cycles nécessaires dépend de la complexité de l'instruction.
CPI = nombre de cycles moyen requis pour effectuer 1 instruction.

Instruction par cycle (IPC)

ICP = nombre moyen d'instructions exécutées pour chaque cycle du signal d'horloge.

Puissance d'un processeur - Métriques

FLoating point Operations Per Second

Nombre d'opérations (additions ou multiplications) à virgule flottante par secondes.
Le **nombre de FLOPS/s** est une mesure commune de la vitesse d'un processeur.

Puissance crête

Elle mesure les performance des unités de calcul en virgule flottante (FPU) contenues dans le cœur.

$$R_{peak} = N_{FLOP/cycle} x Freq x N_c$$

Exemple sur un processeur avec:

- fréquence d'un cœur : 3.2GHz
- nombre de cœurs : 6
- Nombre d'opérations flottantes effectuées par cycle : 4

La performance crête d'un processeur de ce type sera donc : $4 \times 3.2 \times 10^9 \times 6 = 76.8$ GFLOPS/s
En pratique, on sera souvent loin de la puissance théorique (qui dépend aussi de la charge de la machine, du système d'exploitation, ...)

Pour résumer

Un cœur de calcul comprend:

- des unités de calcul : calculs arithmétiques élémentaires sur des nombres entiers, opérations logiques, calculs flottants,
- de la mémoire : registres et cache L1,
- du contrôle pour séquencer le déroulement des calculs.

Si on veut aller plus loin

le cœur inclus beaucoup de mécanismes d'optimisation complexes :

- Pipelining, processeur superscalaire : pour pouvoir traiter plusieurs choses en même temps.
- Exécution spéculative, prédiction de branchement : pour prendre de l'avance sur les instructions du programme.
- Hyperthreading : deux processeurs logiques sur une seule puce, permettant d'utiliser au mieux les ressources du processeur.
- Instructions vectorielles : le processeur, en fonction de son jeu d'instructions, est capable d'appliquer la même instruction simultanément à plusieurs données (SSE, AVX).

Un exemple de parallélisme hardware

Parallélisme d'instruction

Instructions vectorielles : un premier type de parallélisme hardware

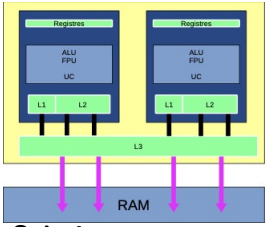
Exemple instruction SSE

```
Algorithme :
for i=1,n
x[i] = sqrt (x[i])
```

```
Travail du processeur :
for i=1,n
load x[i] to the  floating-point register
calculate the square root
write the result from the register to memory

Avec SSE :
for {i1,i2,i3,i4} in {1:n}
load x[i1],x[i2],x[i3],x[i4] to the SSE register
calculate 4 square roots in one operation
write the result from the register to memory
```

Conclusion : hardware mais à garder en tête pour optimisations:

Interaction CPU/RAM - Problématique des accès mémoires



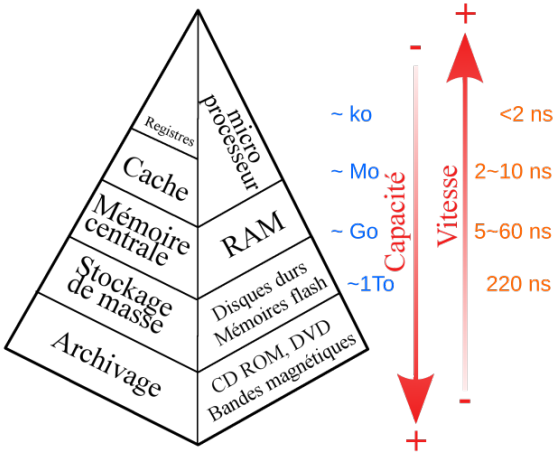
- **Principe de fonctionnement** : le CPU a besoin d'un débit soutenu en lecture d'instructions et de données.
- **Problème** : la RAM qui stocke ces instructions et données est beaucoup trop lente pour assurer ce débit.

Solutions

- Hiérarchie mémoire : utilisation de mémoires intermédiaires entre la RAM et le CPU.
  $\longrightarrow$ permet de stocker les données utiles au plus proche du CPU.
- Localité mémoire. Pour favoriser
  - les accès à des adresses mémoires proches du processeur (registres, cache L1, cache L2)
  - le réemploi de données amenées dans les caches.
    $\longrightarrow$ Permet d'éviter les temps de pause du CPU et de calculer plus vite.

Les mémoires - Hiérarchie mémoire

Capacité vs. vitesse d'accès



Les mémoires - Hiérarchie mémoire

- Les *registres* qui appartiennent au CPU.
  ++ Extrêmement rapides,
  ++ Fonctionnent à la vitesse des CPU
  Capacité très minimale: **qq Ko**
- La *mémoire cache* (L1, L2, L3):
  + Très rapides
  + Mettent à disposition du CPU les copies de quelques données et instructions prises dans la mémoire centrale.
  Capacité: **qq Ko** $\sim$ **128 Mo**
  La cache L1 est le plus rapide mais de taille plus restreinte.
- La *RAM/mémoire centrale*:
  + Vitesse de réaction est trop lente
  + Les données y sont lues par blocs et mises à portée de main du CPU grâce à la mémoire cache.
  Capacité: **1**$\sim$ **jusqu'à quelques To** (très cher !)
- **Mémoire de masse** : les disques (**plusieurs dizaines de To**)

Les mémoires - Fonctionnement et organisation des caches - Localité mémoire

Lorsque le processeur tente d'accéder à une information (instruction ou donnée)

- si l'information se trouve dans le cache (*hit*) : le processeur y accède sans état d'attente,
- si l'information ne s'y trouve pas (*miss*) : le cache est chargé avec un bloc d'informations de la mémoire:
  $\rightarrow$ risque de renvoi de "victime" dans la RAM $\rightarrow$ CPU en attente pendant ce temps.

Les mémoires sont en général conçues pour exploiter les principes de localité mémoire .

- **Localité spatiale :** lorsqu'un programme accède à une donnée ou à une instruction, il est probable qu'il accédera ensuite aux données ou instructions voisines.
- **Localité temporelle :** lorsqu'un programme accède à une donnée ou à une instruction, il est probable qu'il y accédera à nouveau dans un futur proche

Les mémoires - Localité mémoire et bonnes pratiques

```
subroutine sumVec (vec ,n)
integer:: n
integer:: vec(n)
integer:: i,sum=0
do i=1,n
 sum=sum+vec(i)
end do
end subroutine
```

→ Bonne localité spatiale des données du tableau `vec` : *accès en séquence*

→ Bonne localité temporelle de la donnée `sum` : *accès fréquent*

Accès optimal en fortran

```
do j = 1, n
  do i = 1, n
    y(i) = a(i,j) * x(j)
  end do
end do
```

→ Bonne localité spatiale: *stockage des colonnes les unes derrière les autres*

Accès optimal en C

```
for (i=0; i<n; ++i){
  for (j=0; j<n; ++j){
    y[i] += a[i][j] * x[j];
  }}
```

→ Bonne localité spatiale: *stockage des lignes les unes derrière les autres*

Les communications internes et externes

Temps d'accès à la mémoire - Quelques définitions.

- Bus = ensemble de liaisons physiques pouvant être exploitées en commun par plusieurs éléments matériels afin de communiquer.
- Bande passante = débit d'informations ; quantité d'informations échangée par unité de temps.
- Latence = Temps de réponse du bus à une requête de transfert, temps minimum d'établissement de la connexion : indépendant de la quantité de données à transporter.

Quelques outils pour obtenir des infos sur les système ...

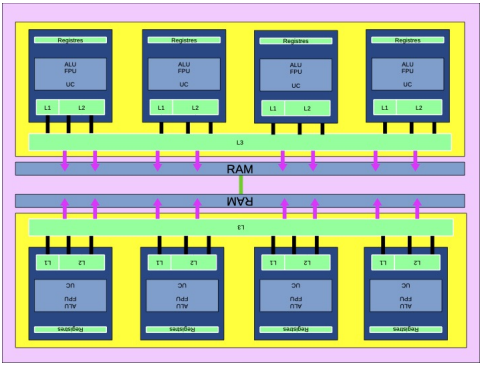
- Informations concernant les “CPUs”

```
>cat /proc/cpuinfo

processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
...
```

- Taille de la RAM

```
>free -m
total        used        free      shared  buff/cache   available
Mem:         128850        1945      115924           4       10980      125861
```

Du processeur au nœud



Nœud = Ressource de calcul contenant plusieurs processeurs (socket) partageant une même mémoire et reliés par un lien QPI.

Les mémoires - Temps d'accès à la mémoire partagée

Deux classes d'architectures parallèles à mémoire partagée différenciées par leur *temps d'accès à la mémoire*.

UMA (Uniform Memory Access)

Même temps d'accès à une zone mémoire, quel que soit le processeur ayant fait la requête.

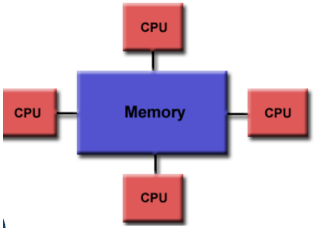
Exemple : systèmes SMP (Symmetric MultiProcessing).

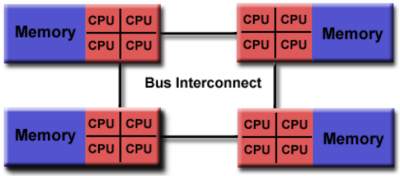
Augmentation nombre de cœurs ⇒ accès concurrents, encombrement ...Perte d'efficacité.

NUMA (Non Uniform Memory Access)

Temps d'accès à la mémoire dépend du processeur.

Plusieurs blocs SMP inter-connectés





Quelques outils pour obtenir des infos sur les système ...

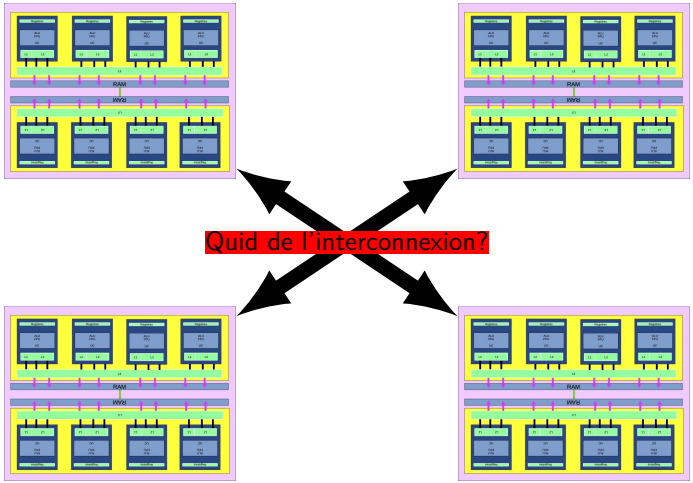
- **hwloc, lstopo** : en ligne de commande pour obtenir des informations sur la topologie du système, les caches etc. Résultat : le type de schéma de la page précédente.
- **lspci** : informations sur les périphériques du ou des bus PCI. Exemple : vérifier la présence d'un GPU

```
> lspci |grep NVIDIA
02:00.0 3D controller: NVIDIA Corporation GK110BGL [Tesla K40m] (rev a1)
```

- Monitoring d'un nœud : **htop**

Du nœud au cluster de calcul

Cluster de calcul : ensemble de plusieurs nœuds reliés par un réseau d'interconnexion.



Du nœud au cluster de calcul - Réseau d'interconnexion

Explosion du nombre de nœuds, cœurs, volume/mouvement de données
$\longrightarrow$ **besoin d'un réseau d'interconnexion rapide pour les communications**

Comment?

- Les nœuds sont stockés dans des **racks**.
- Chaque nœud est connecté à l'aide d'un switch sur le réseau. $\rightarrow$ **Pas de mémoire partagée !**

Caractéristiques des principaux types

Technologie	Latence	Bande passante	Message Rate
10Gb Ethernet	$\sim$ 10 us	10 GB/s	$\sim$ 1 Million msg/sec
IB FDR (Mellanox)	0.7 us	56 GB/s	$\sim$ 35.9 Millionsg /sec
IB EDR (Mellanox)	0.5 us	100 GB/s	$\sim$ 100 Million msg/sec
Omni-PAth (Intel)	0.8 us	100 GB/s	$\sim$ 100 Million msg/sec

Gestion des E/S (ou I/O) [2]

HPC ⟶ Explosion du volume/mouvement de données ⟶ Lecture/Écriture

I/O = Échange d'informations entre le processeur et les périphériques qui lui sont associés.

La **gestion des IO** est rendue possible par l'utilisation d'un *système de fichier*

Rôles:

- Organiser le stockage des données sur un support physique
- Gérer l'espace de nommage et les attributs

Types d'information traitées:

- Données: contenu des fichiers
- Méta-données: ensemble d'informations sur le fichier : nom, position du fichier sur le support physique, taille, attributs (propriétaire, groupe, ...)
- Journal: enregistrement des opérations avant leur exécution pour assurer la cohérence en cas de crash (coupure d'alimentation par exemple).

[2] Entrée/Sortie (ou Input/Output)

Différents types de systèmes de fichiers dans le monde du HPC[3]

- **Systèmes de fichiers séquentiels: Ext4, XFS, BtrFS,...**
  Utilisés en local sur les serveurs et les nœuds, parfois au dessus d'un système plus bas niveau (système RAID) permettant d'exploiter plusieurs disques pour des raisons de fiabilité et de performance.
- **Systèmes de fichiers partagés: NFS,NFSv4+RDMA, SSHFS,...**
  En général une couche réseau au dessus d'un système de fichiers, permettant de s'y connecter par le réseau.
- **Systèmes de fichiers distribués: BeeGFS, Lustre, Ceph,...**
  Partagés par définition, ils parallélisent les performances de plusieurs supports physiques et serveurs en `strippant` les fichiers (blocs distribués sur plusieurs serveurs)
- **Systèmes de fichiers objet: Irods, Swift, rados,...**
  Les objets peuvent être des fichiers, des blocks. Ils sont stockés sur des serveurs différents et les méta-données sont stockées à la manière d'une base de donnée.
  L'interface n'est pas POSIX (on doit généralement faire des `put` et des `get`).

[3] High Parallel Computing

Un cas concret : la machine Dahu

Une seule machine, **plusieurs espaces de stockage avec des caractéristiques différentes**:

- **/home** : dossiers personnels de taille limitée, monté via *NFS*
- **/bettik** : scratch distribué haute performance sur technologie *BeeGFS*
- **Plateforme Mantis** : stockage de type cloud basé sur la technologie *IRODS*

Un conseil essentiel

A éviter à tout prix
Sources de pertes de performances

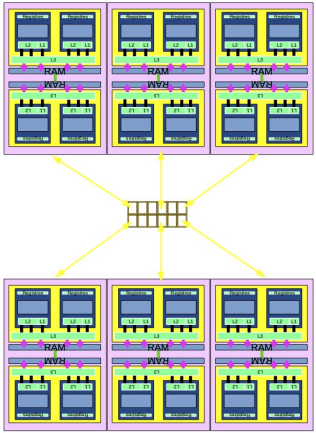
BEAUCOUP de PETITS fichiers! $\longrightarrow$ Saturation du/des serveur(s) de métadonnées

- Un système de fichiers qui gère beaucoup de petits fichiers va passer plus de temps à gérer les meta-données et le journal que les données proprement dites!
- Le nombre et la taille dépendent de la configuration, mais on peut généralement considérer que 10000 fichiers de quelques k-octets peuvent déja poser des problèmes.

Les accès aléatoires ou concurrents sur des petits blocks peuvent poser problème!

- Gestion des “locks” (Conflits d'accès sur la même portion de fichier).
- Préférer les écritures et lectures séquentielles.

Résumé



- Cœur : ressource qui fournit la puissance de calcul,
- Processeur/Socket/CPU : composé d'un ou de plusieurs **cœurs**,
- Nœud : ressource contenant plusieurs processeurs qui partagent une même mémoire (NUMA).
- Réseau rapide IB, OPA,...: technologie qui relie les nœuds stockés dans des **racks**
- Filesystem : IO
- Frontale, OAR,

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

5 Comment paralléliser ?

6 Compléments

7 Conclusion

Taxonomie des architectures

A ce stade vous aurez compris que ce qui fera la différence entre les types d'architecture sera:

- La manière de distribuer et d'accèder aux données.
- Le *nombre* et la puissance des unités de calcul, des coeurs.
- La qualité des différentes voies de communication.

Une première aide pour paralléliser : classer les architectures en fonction de ces critères.

- Le traitement du **flot d'instructions** : le processeur peut-il exécuter plusieurs instructions simultanées ? Autorise un *parallélisme d'instructions*.
- Le traitement du **flot de données**. Un *parallélisme de données* est-il possible ?
- Le **type de mémoire** : partagée, distribuée ou hybride.
- Le type de réseau disponible : comment faire transiter les données, les messages ?

Taxonomie des architectures

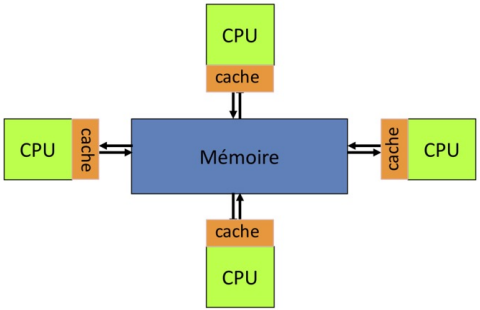
Un point clé : la distribution des données.

1. Mémoire partagée : un pool mémoire commun à tous les processeurs
2. Mémoire distribuée : des zones mémoire exclusives, accessibles uniquement à certains processeurs.

Types d'architectures très différents $\Rightarrow$ modèles de programmation parallèle différents.

Système à mémoire partagée

Un espace mémoire global, accessible simultanément par plusieurs CPUs et donc par plusieurs programmes.



- *Tous les CPUs accèdent à toute la mémoire globale*, avec un même espace d'adressage.
  Chaque CPU est autonome mais les modifications effectuées dans la mémoire partagée sont visibles par tous.
- Les processeurs ont leur propre *mémoire locale* (cache, ...) dans laquelle sera copiée une partie de la mémoire globale.

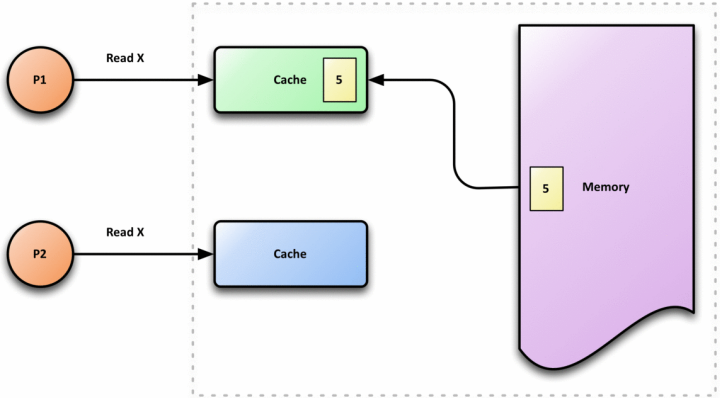
Système à mémoire partagée

Quelques remarques pour anticiper les points forts et faibles des modèles de programmation basés sur ce type de système.

- Accès rapides à la mémoire et partage des données
- Accès concurrents, risque d'embouteillage
- Cohérence des caches ? (Voir ci-après) répercution des écritures faites par un processeur dans son cache vers les autres processeurs.
- Limites de ces systèmes = taille de la mémoire et nombre de CPUs disponibles.
  Parallélisme sur un seul noeud !

Cohérence de cache

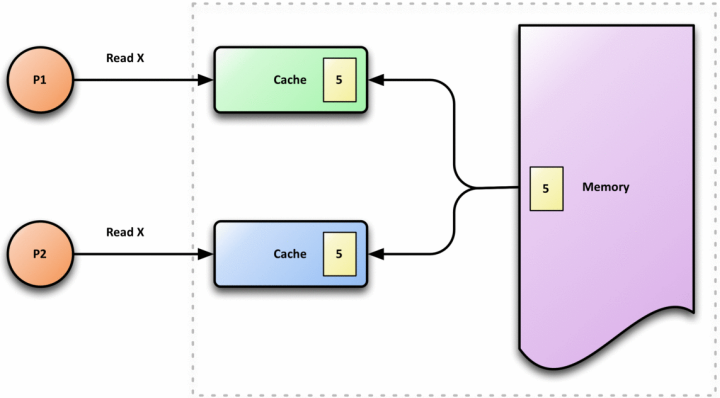
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

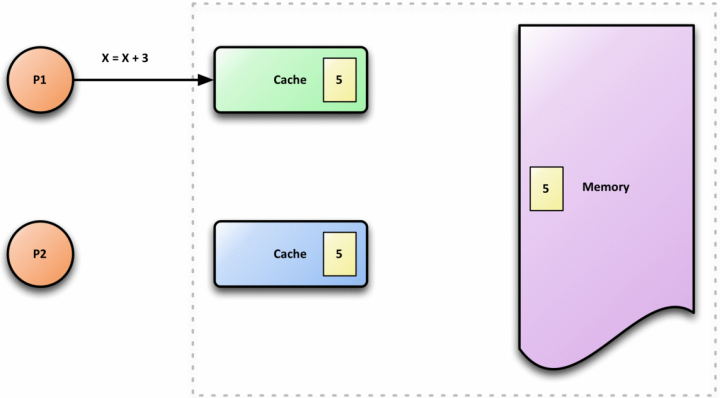
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

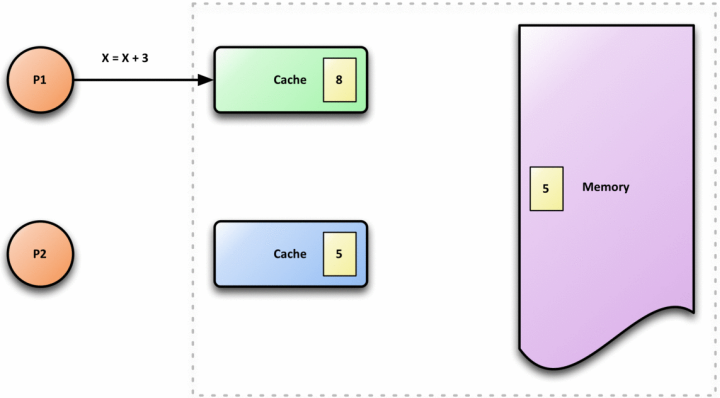
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

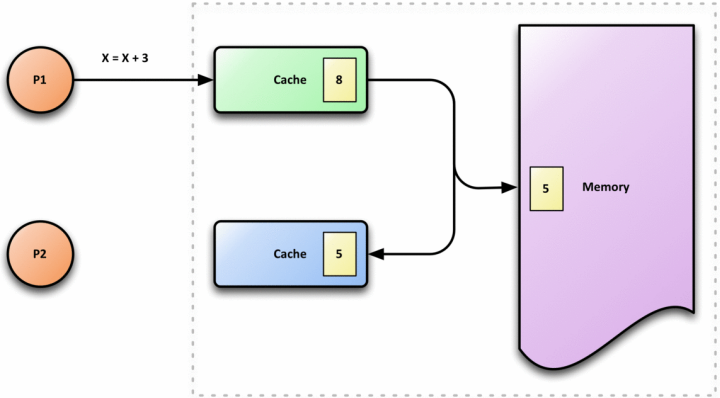
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

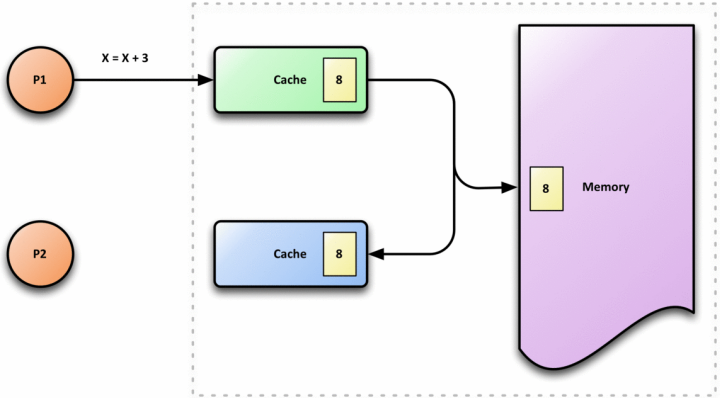
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

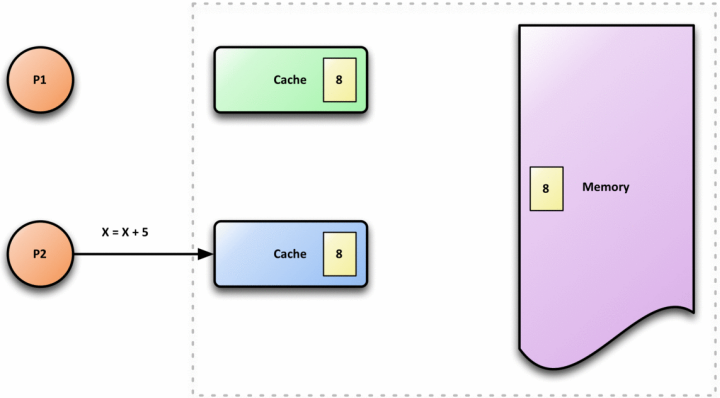
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

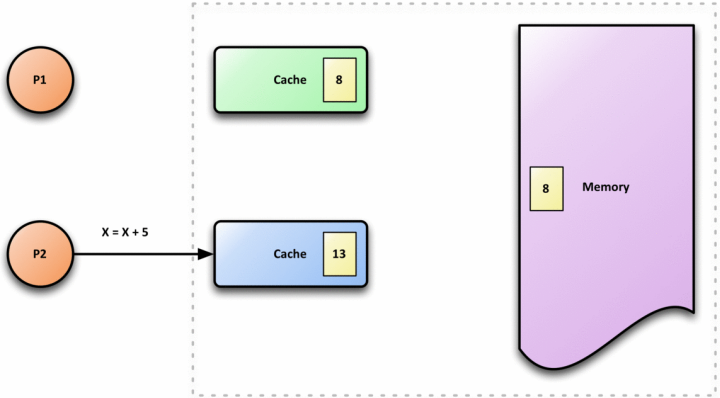
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

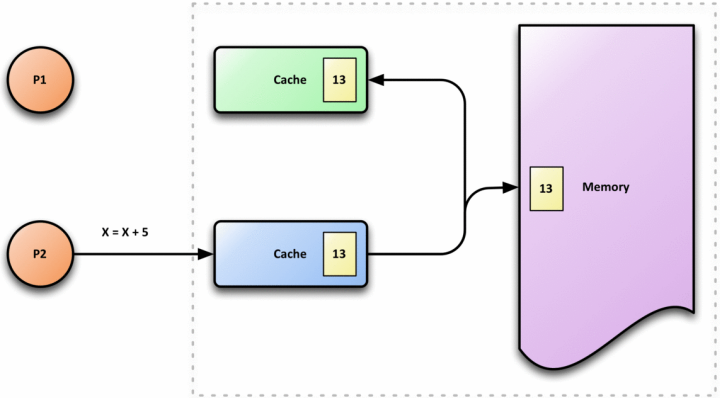
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

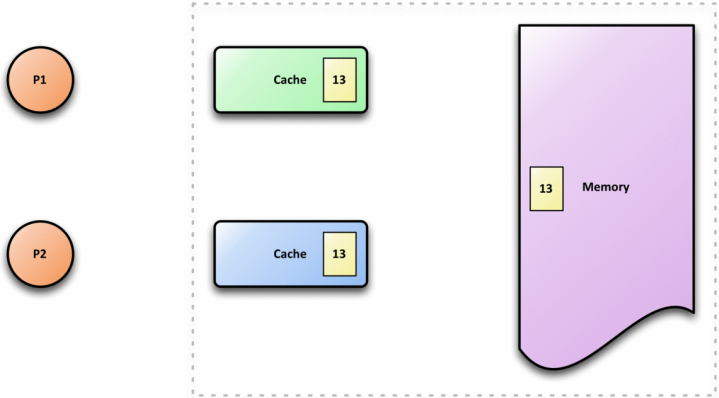
Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Cohérence de cache

Comment est gérée la cohérence de données modifiées par plusieurs processeurs ?



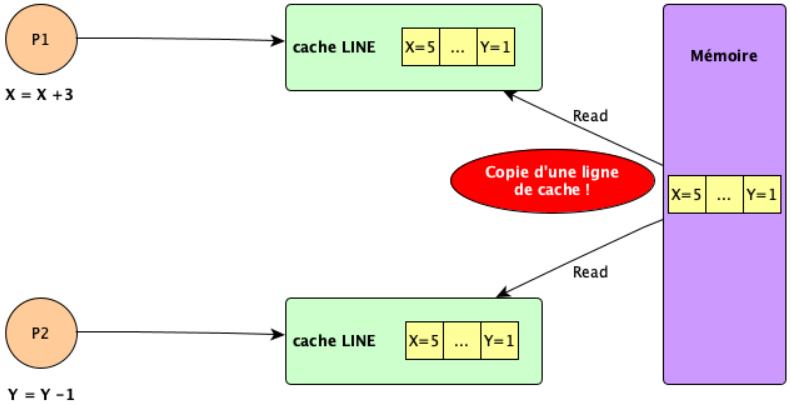
Sources : by M3tainfo - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=52362567

Gestion de la cohérence de cache

Attention : cette gestion a un coût !
Risque de false sharing, une dégradation des performances due à la gestion de la cohérence des caches.
Le false sharing est une conséquence du principe de localité introduit précédemment.
Principe de localité : si une donnée (zone mémoire) est utilisée, elle a des chances d'être référencée à nouveau dans un futur proche (localité temporelle) et les zones voisines ont des chances d'être également utilisées (localité spatiale).
Ce principe est notamment utilisé pour optimiser l'utilisation des mémoires cache/register. Ainsi lorsqu'une donnée est nécessaire pour un processeur, elle sera chargée dans la mémoire cache avec les données voisines ("ligne de cache").

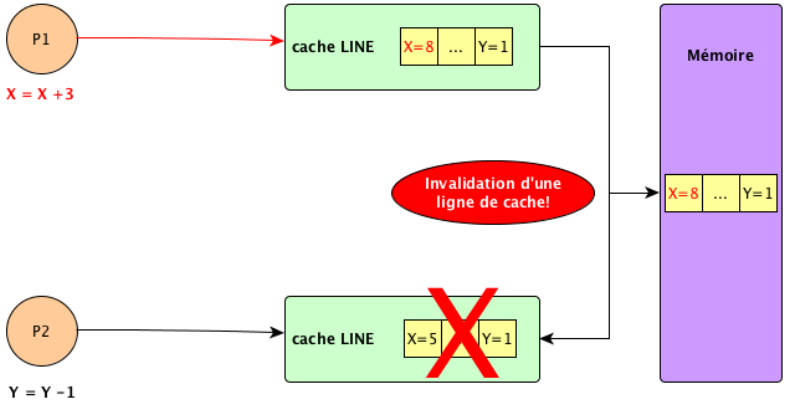
False sharing

False sharing : dégradation des performances due à la gestion de la cohérence des caches.



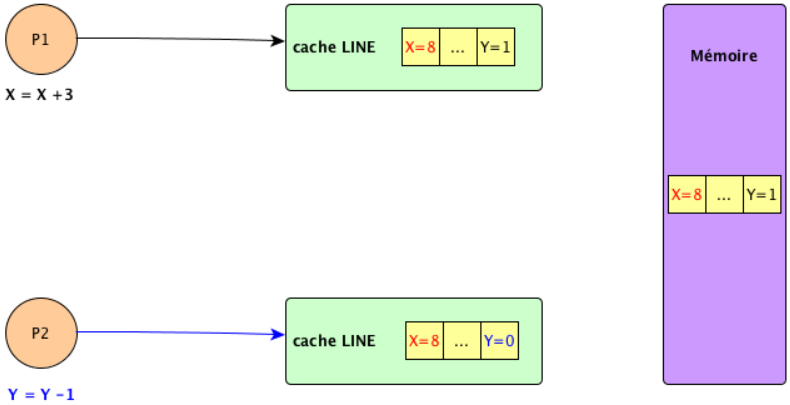
False sharing

False sharing : dégradation des performances due à la gestion de la cohérence des caches.



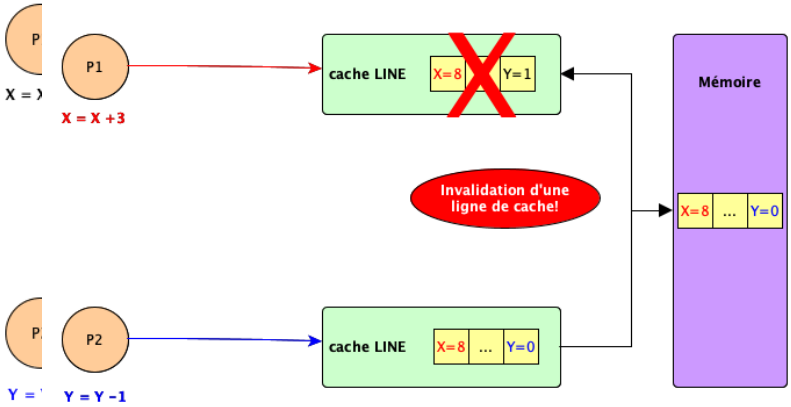
False sharing

False sharing : dégradation des performances due à la gestion de la cohérence des caches.



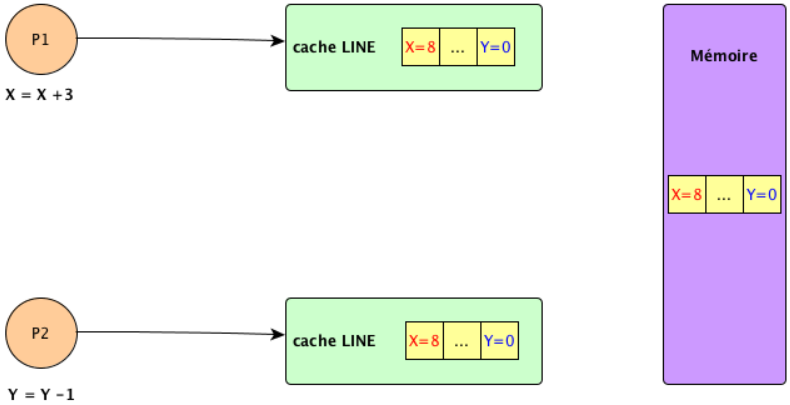
False sharing

False sharing : dégradation des performances due à la gestion de la cohérence des caches.



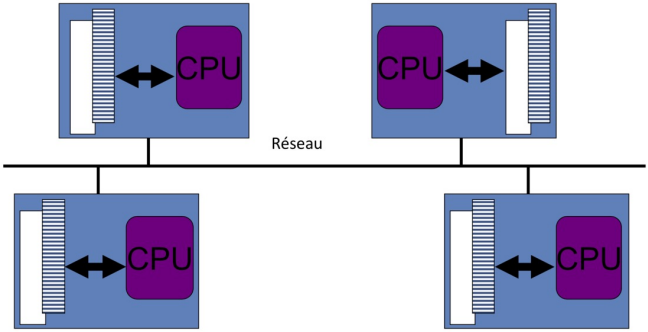
False sharing

False sharing : dégradation des performances due à la gestion de la cohérence des caches.



Systèmes à mémoire distribuée

Système multi-processeurs où la mémoire est répartie sur plusieurs **noeuds**. En conséquence, chaque processeur n'a accès qu'à un sous ensemble de la mémoire. L'accès (lecture ou écriture) à certaines parties de la mémoire nécessite des communications/échanges entre processeurs.



Point clé : un *réseau d'interconnexion* (infiniband ...) pour communiquer entre les mémoires des différents processeurs.

Comparaisons ...

Une zone mémoire commune ?

Mémoire partagée	**Mémoire distribuée**
Oui !	Non !
Facilite la programmation, partage des données entre les tâches rapide.	*Pas d'espace d'adressage global commun à tous les procs. Le programmeur devra gérer explicitement les communications.*
Encombrement, accès concurrents à la mémoire, risque de false sharing.	Accès rapide aux mémoires locales (pas nécessairement de surcoût lié à la gestion de la cohérence des caches)
Gestion de la synchronisation par le programmeur.	

Comparaisons ...

Taille et coût du système ?

Mémoire partagée

- *Très coûteux de construire des architectures à mémoire partagée avec beaucoup de CPUs*

Mémoire distribuée

- Accès à plus de ressources. La mémoire totale disponible peut augmenter proportionnellement avec le nombre de processeurs.
- Coût réduit, plus facile à construire
- Les procs opèrent indépendamment les uns des autres.
  Programmation plus complexe : risques d'erreurs, complexité des algorithmes, gestion des données distribuées ...
- Les performances vont fortement dépendre du réseau d'interconnexion (*potentiellement coûteux*).

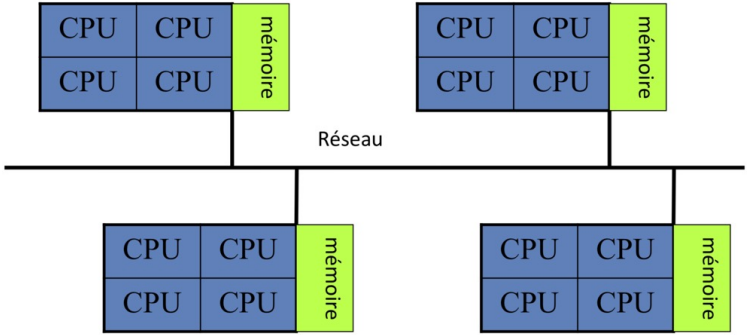
Accélérateurs

Principe : *chargement d'une partie du programme sur un support (co-processeur, device) spécialisé, capable de traiter plus efficacement la partie ciblée.* On parlera d"'offloading" entre le CPU (**host**) et l'accélérateur (**device**).

GPUs, Graphics Processing Unit

A noter que ce type d'infrastructure est très utilisé pour le machine learning et le deep learning.

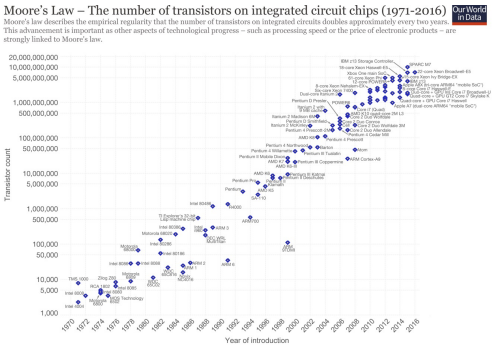
Systèmes hybrides, hétérogènes



La réalité aujourd'hui : un assemblage de briques "mémoire partagée" interconnectées, auxquelles peuvent éventuellement s'ajouter des accélérateurs.

Evolution des architectures, loi de Moore

"Prédiction" de Moore (1971, Intel) : le nombre de transistors des microprocesseurs sur une puce de silicium double tous les deux ans.



Conséquences :

- *Hz-MIPS* ↑ ⇒ *performance des codes* ↑
  gratuitement, simplement en utilisant une machine plus récente.
  Mais ...
- *Hz-MIPS* ↑ ⇒ *puissance* ($T°$!) ↑
  Aujourd'hui les constructeurs favorisent les systèmes avec **plus de coeurs** et **moins de puissance** par coeur.
  ⇒ Rendement (MIPS/Watt) meilleur.
  ⇒ Votre code ne s'améliorera plus tout seul (et sera peut-être même plus lent) !

Evolution des architectures, le top 500

Top 500 : état des lieux, classement des 500 "machines" les plus puissantes au monde.

https://www.top500.org.

Classement établi selon les résultats obtenus au passage de benchmarks (linpack).

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

Evolution des architectures : kilo, méga, giga, téra, péta, ...

Top 500	Puissance en Gflops	Nombre de coeurs	Puissance (kW)
Juin 1993	131	104	?
Juin 2008	1 375 780	122 400	2345
Novembre 2022	1 102 000 000	8 730 112	21100😦
Dahu (gricad)	248000 (406000)	6992	55 (80)

1997 : année du *Teraflops* (10^{12})

↓

2008 : année du *Petaflops* (10^{15})

↓

2022 : vers l'*Exaflops* (10^{18})

Evolution des architectures, quelles tendances ?

Quelques commentaires

- Sortie du classement du numéro 1 : $\approx$ 8 ans.
- Complexité croissante (manycores, machines hétérogènes deviennent majoritaires ...).
- Consommation énergétique (très !) conséquente.

Exemple

Une machine de l'IDRIS : Jean Zay

- Machine dite *convergée* : noeuds hybrides CPU - GPU
- A la fois pour le HPC et le Deep Learning
- Performance crête : 14 PFLop/s
- 61120 coeurs CPU, 1044 GPU

Evolution des architectures, le green 500

https://www.top500.org/lists/green500/.

Rank	TOP500 Rank	System	Cores	Rmax (PFlop/s)	Power (kW)	Energy Efficiency (GFlops/watts)
1	405	**Henri** - Lenovo ThinkSystem SR670 V2, Intel Xeon Platinum 8362 2800Mhz [32C], NVIDIA H100 80GB PCIe, Infiniband HDR, Lenovo Flatiron Institute United States	5,920	2.04	31	65.091
2	32	**Frontier TDS** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	120,832	19.20	309	62.684
3	11	**Adastra** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES) France	319,072	46.10	921	58.021
4	15	**Setonix – GPU** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Pawsey Supercomputing Centre, Kensington, Western Australia Australia	181,248	27.16	477	56.983
5	68	**Dardel GPU** - HPE Cray EX235a, AMD	52,864	8.26	146	56.491

Un classement équivalent mais avec un critère de classement différent : le nombre de *Flops par Watts* consommées.

Calcul et responsabilité sociétale et environnementale (RSE)

Calcul haute performance, supercalculateurs ⇒ consommation énergétique très/trop élevée

- Frontier, Oak Ridge : 8 730 112 coeurs, 21 100 kW
- TGV : 512 sièges, 8800 kW
- Réacteur nucléaire : entre 500 et 1500 MW

Le coût de la consommation est maintenant du même ordre que celui de l'achat de matériel
Il faut donc (au minimum) :

- s'interroger sur la pertinence de l'utilisation d'un gros calculateur pour ses simulations;
- optimiser l'utilisation des ressources : rôle d'un gestionnaire du type oar (exemple : arrêt des noeuds après une certaine période d'inactivité, réveil si ressources allouées ...);
- sensibiliser et informer les utilisateurs (Accounting : coût financier/conso par projet, mise en place de métrique en temps réel pour l'aide à la mesure de l'intérêt des résultats ...).

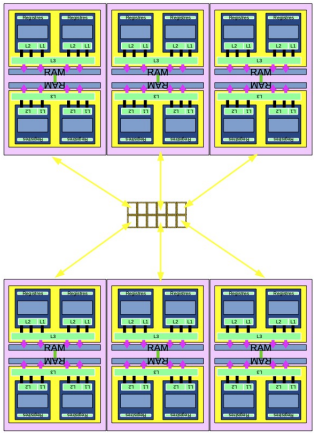
Calcul et responsabilité sociétale et environnementale (RSE)

- GDR Labo1point5 : réduire l'empreinte de nos activités de recherche sur l'environnement *https://labos1point5.org/*
- Le groupe Ecoinfo : réduire les impacts environnementaux et sociétaux négatifs des technologies du numérique. *https://ecoinfo.cnrs.fr*

Que retenir de tout ça ?

Architectures

- La fréquence d'horloge des processeurs stagne, voire diminue.
- La tendance est à la multiplication des coeurs de calcul.
- Les systèmes hétérogènes (CPUs + GPUs ...) deviennent la norme.
- Mémoires hiérarchiques et hétérogènes. Evolution moins rapide que celle des processeurs.
  Point crucial : l'accès aux données, les transferts en mémoire.
  Le coût des transferts est de + en + élevé par rapport aux temps de calcul.



Du parallélisme partout (y-compris dans votre portable !) : difficile/impossible de passer à côté.

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

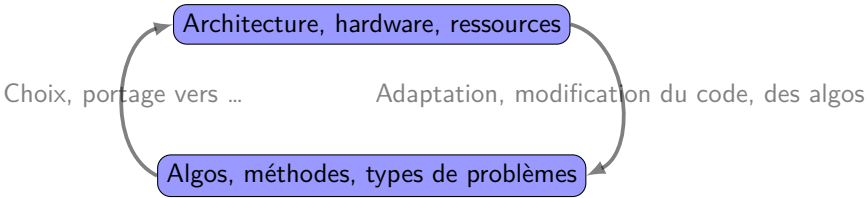
5 Comment paralléliser ?

6 Compléments

7 Conclusion

Comment paralléliser?

- ✓ *Connaître/comprendre l'architecture* des ressources disponibles.
- *Choisir un modèle de programmation adapté.*
- *Connaître et comprendre le comportement du programme*, des algorithmes.



Quel modèle pour quel type de parallélisme ?

A chaque type d'architecture parallèle correspondent un ou plusieurs paradigmes de programmation, avec les langages associés. Difficile de faire un choix et de s'y retrouver ...
Voici les associations types d'architecture/modèles/langages classiques :

- Inter-noeuds par échange de messages, systèmes à mémoire distribuée ⇒ MPI.
  - Accès à plus de ressources : il suffit d'ajouter des noeuds.
  - Points critiques : les **communications**, la **distribution des données**.
- Intra-noeud, systèmes à mémoire partagée, processus légers (threads) ⇒ OpenMP.
  - Programmation plus facile, pas à pas.
  - Espace d'adressage global, mémoires rapides, synchronisations à gérer dans le programme
  - Limites : augmentation du nombre de CPUs sur un noeud ⇒ encombrement des accès à la mémoire
- SIMD, accélérateurs ⇒ OpenMP, OpenACC, OpenCL, Cuda.
- Systèmes hybrides/hétérogènes : combinaisons de plusieurs parallélisme/modèles (MPI + OpenMP + GPUs ...)

Quel modèle pour quel type de parallélisme ?

Quelques références.
MPI (Message Passing Interface). Voir cours plus loin.
OpenMP (Open Multi-Processing). Voir cours plus loin.
OpenACC (Open Accelerators, *https://www.openacc.org*),
OpenCL (Open Computing language, *https://www.khronos.org/opencl/*) : standards conçus pour le calcul parallèle sur systèmes hétérogènes (avec accélérateurs).
Cuda. Voir cours plus loin.

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

5 Comment paralléliser ?

6 Compléments

7 Conclusion

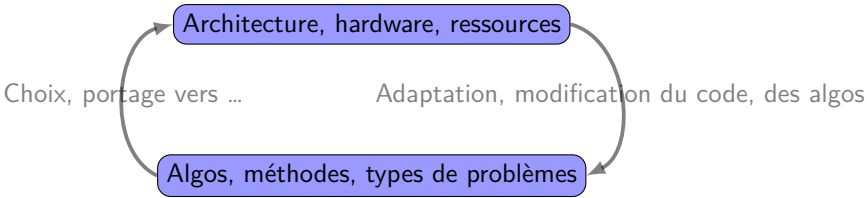
Quels bénéfices attendre de la parallélisation?

Rappels (voir intro) de nos objectifs ...
Améliorer les performances
Mieux (?) exploiter les ressources
Et pour répondre à ces besoins :

- ✓ *Comprendre et connaître l'architecture* des ressources disponibles, pour les choisir correctement et les exploiter pleinement.
- ✓ *Connaître les modèles de programmation* parallèle ET leurs liens avec les architectures.
- *Comprendre le comportement du programme*, des algorithmes.

Nous aurons besoin d'*évaluer les performances* du calcul, de *métriques, d'indicateurs* ...

Comprendre le programme, évaluer ses performances



- *Evaluer les performances* d'un programme dans différents contextes (en séquentiel, parallèle, accéléré ...).
- Trouver des *indicateurs pertinents*, des métriques. Savoir les interpréter.
- *Calculer le gain* relativement à un calcul séquentiel ou sur une autre architecture.

Rappel : pour les outils de profiling en séquentiel, voir le module "Outils pour le développement et l'utilisation de logiciels de calcul scientifique".

Speedup

Un premier indicateur de performance : *la comparaison des temps d'exécution*.
Soit T_{ref} un temps de référence et T_p le temps de calcul sur l'architecture à évaluer. On définit l'accélération relative (*speedup*) comme :

$$S = \frac{T_{ref}}{T_p}$$

Remarques :

- *S et T_p vont bien entendu dépendre des ressources impliquées, par exemple du nombre de coeurs utilisés.*
- *T_{ref} n'est pas nécessairement le temps en séquentiel. On peut vouloir par exemple obtenir l'accélération d'un calcul sur un GPU, relativement au même calcul sur plusieurs CPUs.*

Speedup

A retenir :
quels que soient les efforts, le speedup est toujours limité par la partie de code non parallélisable

Et donc, une première étape vers la parallélisation :

- Identifier les parties coûteuses du code séquentiel (profiling).
- Identifier les parties pouvant être accélérées (découpées, distribuées, réorganisées, bref parallélisables !)

⇒ Une première idée du gain possible ...et de l'intérêt des efforts à fournir.

Efficacité

On définit *l'efficacité* comme le rapport entre l'accélération globale (le speedup) et celle de la partie optimisée (la fraction de code parallélisable).

$$efficiency = \frac{S(n)}{n}$$

L'efficacité fournit une sorte de *rendement du calcul parallèle* : une valeur de 1 signifie que 100% des ressources ont été utilisées pour du calcul utile.
En général :

$$0 \leq Speedup(n) \leq n, \quad 0 \leq Efficacite \leq 1$$

Cas particuliers

- **speedup/efficacité lineaire** : $e = 1, S(n) = n$
- **superlinear speedup/efficiency** : $e > 1, S(n) > n$.
  Une explication possible : effets de cache. Plus de ressources $\Rightarrow$ plus de données stockées dans des mémoires rapides.

Load Balancing/équilibrage de charge

Mesure de la qualité de la répartition du travail entre les ressources.

C'est une notion importante : *le temps de calcul sur la ressource la plus lente détermine le temps de l'application.*

Que faire?

- **Attention à la distribution des données!**
  Si elle est statique (ne change pas au cours de la simulation) on peut facilement faire le nécessaire pour que chaque ressource traite un volume équivalent.
  Si elle est dynamique et imprévisible il faudra éventuellement utiliser un algo de *dynamic load balancing*, potentiellement coûteux ...
- **Attention à la répartition des tâches** (boucles etc)
- Prendre en compte les différences de capacité des différentes ressources (e.g. un noeud avec ou sans GPU, des processeurs différents etc)
- Minimiser les temps d'attente (recouvrement des comms .... Voir cours MPI).

Scaling

Analyse de la réponse du système lorsqu'on augmente les ressources.

- *Strong scaling* : évolution du temps de calcul avec le nombre de ressources pour un **problème de taille fixe**. Exemple : évolution du temps de calcul en fonction du nombre de coeurs utilisés (i.e. le speedup)
- *Weak scaling* : évolution du temps de calcul avec le nombre de ressources pour un problème dont la taille **par ressource** est fixe. Exemple : "scaled speedup". $T_{ref}(N)$ = Temps de référence = temps pour résoudre un problème de taille N sur un coeur. $T_p(n, n * N)$ = temps pour résoudre un problème de taille N * n sur n coeurs.

$$\hat{S}(n) = \frac{T_{ref}(N)}{T_p(n, n * N)}$$

Si $\hat{S}(n)$ est proche de 1, alors le code est apte à gérer la montée en charge. On parlera de *scalability : capacité du système à traiter un volume de travail croissant lorsqu'on augmente les ressources.*

Les indicateurs - Floating-point Instructions et intensité arithmétique

La majorité des applications scientifiques sont dominées par des opérations d'arithmétique flottante.

FLOPS/s - FLoatting point Operation Per Second

Unité de mesure pour les calculs mathématiques en nombres flottants.
Le **nombre de FLOPS/s** est une mesure commune de la vitesse d'un processeur.

Intensité arithmétique - IA

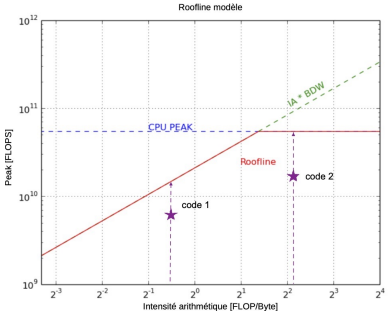
Ratio entre le nombre d'**opérations sur les nombre flottants** (FLOPS) et le volume de données transférées (en Bytes) pendant l'exécution [FLOPS/Byte].

```
IA = #FLOP / #Bytes
```

Rappel : l'accès aux données, leurs déplacements en mémoire est coûteux. Une IA élevée signifie une utilisation optimale des données du point de vue du coût en mémoire.

Les indicateurs - Roofline model

Pourquoi ? proposer une méthode 'visuelle' pour aider à mieux prévoir le comportement d'un code et les optimisations possibles



Le roofline model est un graphe 2D avec :

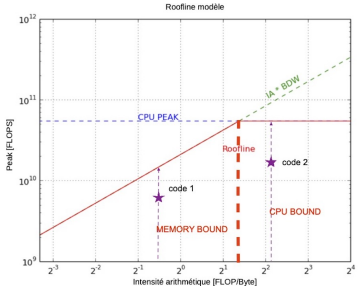
- l'IA du code [FLOPS/Byte] en abscisse,
- la performance atteignable [FLOPS/s] en ordonnée.

Performance = f(Peak, Bande passante, IA)

Peak: limite de performance théorique de la machine [FLOPS]

Les indicateurs - Roofline model

Pourquoi ? proposer une méthode 'visuelle' pour aider à mieux prévoir le comportement d'un code et les optimisations possibles



- **Memory bound**: La bande passante mémoire limite la vitesse de traitement, le CPU n'est pas saturé. Il faut augmenter l'IA.
  La relation entre l'IA et la vitesse de traitement est linéaire et ne dépend que de la bande passante utile.
- **CPU bound** : l'IA a atteint une valeur suffisante pour saturer le CPU, un palier est atteint. Il correspond au pic utile du processeur.

Ref : https://people.eecs.berkeley.edu/~kubitron/cs252/handouts/papers/RooflineVyNoYellow.pdf

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

5 Comment paralléliser ?

6 Compléments

7 Conclusion

Compléments : outils d'analyse et debug

- Scalasca, *http://www.scalasca.org* : analyse de performance et optimisation de code parallèle.
- Debuggers : DDT, *https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt*, Totalview, *https://www.roguewave.com/products-services/totalview*.

1 Contexte et présentation du module

2 Avant de paralléliser : connaitre et comprendre les ressources

3 Quelles architectures aujourd'hui ?

4 Les modèles de programmation parallèle

5 Comment paralléliser ?

6 Compléments

7 Conclusion

Conclusion - Architectures, modèles. Quelles conséquences pour les développeurs?

Parallélisme (quasi) inévitable à tous les niveaux : du portable au supercalculateur.

Importance de la **compréhension de l'architectures des machines** et des modèles de programmation parallèle.

Architecture ⇒ Modèle ⇒ Soft

- *Adapter* les algorithmes, les méthodes numériques, la programmation.
  e.g. algorithmes générant un grand nombre de tâches indépendantes.
- *Appréhender* les évolutions : proposer des codes pérennes, capables d'évoluer, portables.
  Attention à ne pas trop spécialiser/optimiser son code pour une seule architecture!

Soft ⇒ Modèle ⇒ Architecture

- *Comprendre et analyser* les programmes.
- *Choisir* les infrastructures de calcul adaptées.

Conclusion - Architectures, modèles. Quelles conséquences pour les développeurs?

L'utilisation de super calculateurs a un coût environnemental très important .

Il est donc fondamental, au minimum, de

- toujours s'interroger sur la pertinence, la finalité des calculs. Sont ils vraiment nécessaires ?
- d'optimiser l'utilisation des ressources et donc de bien connaître les infras et les outils à disposition.

Conclusion - Conséquences pour les codes de calcul

- Programmation complexe, modèle potentiellement intrusif (MPI...).
- Besoin d'outils/environnements de programmation adaptés.
- Vers des modes de programmation hybrides (MPI + OpenMP + GPUs ...)

Points clés et questions à se poser.

- Identifier des *tâches indépendantes*.
- Identifier les *zones "parallélisables"*.
- La répartition/*distribution des données*. Correspondance avec l'architecture mémoire? Le modèle? Vers le CPU? Vers un accélérateur?
- Les communications.
- Evaluer/profiler le code (séquentiel puis parallèle), identifier les zones "coûteuses".

Calcul et contexte grenoblois

Céline Acary-Robert, Frédéric Audra, Franck Pérignon

février-mars 2023

Accès et utilisation des ressources grenobloises

1. *Contexte européen*
   - Généralités
   - Les centres nationaux
   - Les ressources locales

2. *Accès aux plateformes de calcul et leur utilisation*
   - Présentation des ressources locales
   - Gestion de l'environnement logiciel
   - Gestion des jobs sur les plateformes
   - Conclusions

1 *Contexte européen*
- Généralités
- Les centres nationaux
- Les ressources locales

2 *Accès aux plateformes de calcul et leur utilisation*

Généralités - Le calcul en Europe.

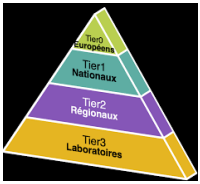
Nombreuses thématiques

- Autour de la recherche fondamentale,
- Autour de la recherche appliquée

Mutualisation des moyens

- Autour du calcul
- Autour du stockage de données
- Autour de la formation

Pyramide du calcul intensif:



Tier	Nbre cœurs	Stockage	Puissance
0	>120 000	30 Po	> 10 Pflops
1	80 000	15 Po	1-10 Pflops
2	10 000	2 Po	< 1 Pflops
3	1 000	500 To	< 400 Tflops

Généralités - Les centres nationaux

IDRIS : Institut du développement et des ressources en informatique scientifique

- Centre de ressources informatiques et pôle de compétences en HPC et IA du CNRS

CINES : Centre Informatique National de l'Enseignement Supérieur

- Etablissement Public à caractère Administratif national du MESRI

Généralités - Les centres nationaux

- Leur rôle est de proposer :
  - des moyens en calcul numérique intensif
  - des solutions d'archivage pérenne de données électroniques (CINES)
  - l'hébergement de plates-formes informatiques d'envergure nationale
- Ils participent aux projets européens
  - PRACE
  - EOSC
  - Ces projets ont pour mission de structurer les offres autour du HPC (matériels, **formations**, solutions logicielles, etc ...) : listes de diffusion

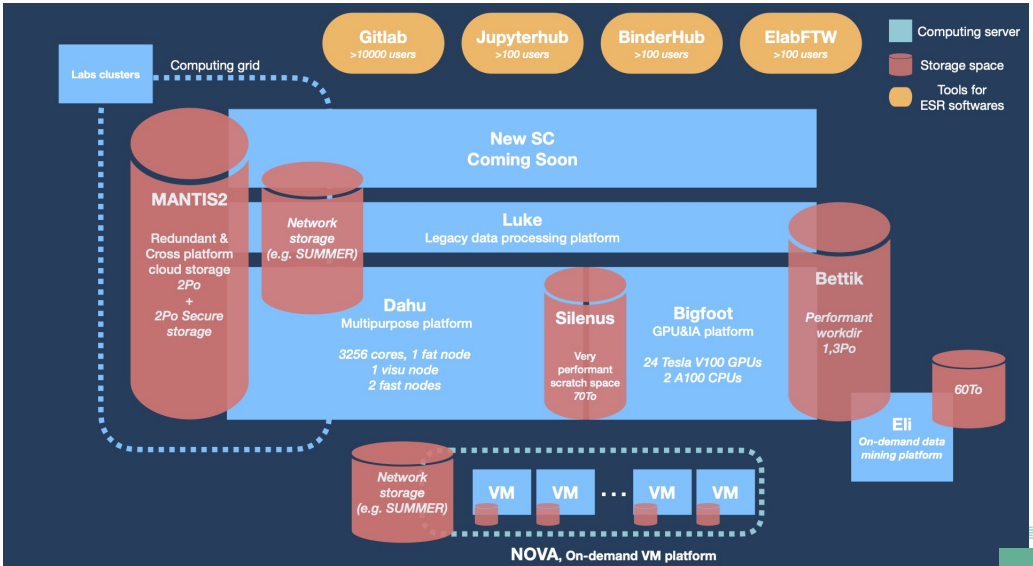
Les ressources locales : GRICAD

L'Unité d'Appui à la Recherche **Gricad** Grenoble Alpes Recherche - Infrastructure de Calcul Intensif et de Données fournit à la communauté enseignement-supérieur / recherche à Grenoble :

- un **accès à des plateformes de calcul**, de grille et de stockage associées,
- un **accompagnement et conseils dans le périmètre du calcul et du traitement de la donnée** pour l'utilisation de ces plateformes.

Documentation : https://gricad-doc.univ-grenoble-alpes.fr/

Les plateformes de calcul de GRICAD



1 Contexte européen

2 Accès aux plateformes de calcul et leur utilisation

- Présentation des ressources locales
- Gestion de l'environnement logiciel
- Gestion des jobs sur les plateformes
- Conclusions

Plateforme Dahu

Particulièrement adaptée au calcul parallèle

- Machine mise en production en décembre 2018
- Adaptée au calcul parallèle
- Évolutive par ajout de nœuds
- 6992 coeurs
  - 4192 Xeon SKL Gold 6130 @ 2.10GHz
  - 2048 Xeon SKL Gold 5218 @ 2.30GHz
  - 432 Xeon SKL Gold 6126 @ 2.60GHz
  - 256 Xeon SKL Gold 6244 @ 3.60GHz
  - 64 Xeon SKL Silver 4216 @ 2.1GHz
  - Réseau Omnipath 100Gb

Supercalculateur: Système informatique très puissant utiliser pour des applications scientifiques complexes (calculs numériques intensifs)

Plateforme Bigfoot

Cluster réservé au GPGPU (calcul sur GPU)

- Machine mise en production en 2022, évolutive par ajout de nœuds.
- Réservé au calcul sur GPU
- 1152 coeurs + 70 coeurs (noeuds 'virgo*' disponibles sur bigfoot entre 00h et 06h.)
- 4 types de noeuds GPU, pour un total de 38 GPUs :
  - 4 x NVIDIA V100 32GB Mémoire : 48GB de RAM par GPU, interconnexion SXM2 (NVLINK) 300GB/s.
  - 2 X NVIDIA A100 40GB Mémoire : 96GB de RAM par GPU, interconnexion PCIe (64GB/s).
  - 2 X NVIDIA A100 40GB Mémoire : partitionnées en 2x7 MIG (Multi-Instances Gpu).
  - 1 x NVIDIA T4 sur les noeuds virtuels virgo*.

Cluster: Regroupement de serveurs en une infrastructure virtuelle unique.

Description détaillée dans la documentation Gricad.

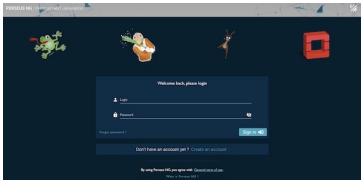
Autre plateforme disponible

Luke - *Dédiée aux calculs séquentiels ou traitement de données massives*

- Machine de 48 noeuds hétérogènes entre 16 et 72 coeurs par noeud, jusqu'à 512GB RAM par noeud.
- Dédiée aux calculs séquentiels ou traitement de données massives
- Partitions dédiées aux projets

L'accès aux plateformes - PERSEUS

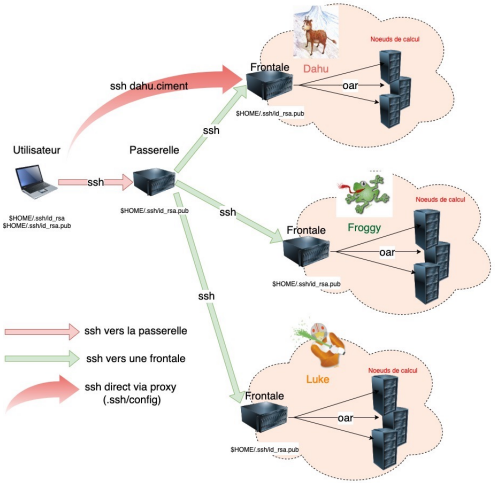
Point d'entrée unique aux services : https://perseus.univ-grenoble-alpes.fr/



- Demande de création de **compte**
- Création/rattachement **projet**
- **Support**: ticket, liste de diff
- Gestion de projets: **wiki**
- Statut du compte/projet
- Informations sur l'activité (consommation)
- Abonnement listes de diffusion

Validation des projets par les responsables scientifiques et techniques de chaque communauté

Accès aux clusters : connexion SSH



- Passerelle : porte d'entrées obligatoire pour tous les clusters : **access-gricad.u-ga.fr** (aucune activité)
- Frontale : porte d'entrée du cluster visé (développement, compilation, **pas de calcul**)
- Noeuds de calcul : oui, là vous pouvez calculer !
- Connexion SSH via les bastions
- Connexion directe après configuration proxy

Utilisation de gestionnaires d'environnement logiciel

Une fois connecté sur les machines de Gricad, vous allez mettre en place votre environnement logiciel ie installer les logiciels dont vous avez besoin.

Définition

Gestionnaires d'environnement logiciel : un outil logiciel qui permet à un utilisateur d'installer dans sa session tous les logiciels dont il a besoin

Caractéristiques

- Fonctionne un peu comme un gestionnaire de paquets (apt, yum, ...) mais propre à chaque utilisateur
- Plusieurs gestionnaires d'environnement logiciels sont installés sur les machines de Gricad : **Nix**, **Guix**, **Conda**, **Singularity**, ...
- Il existe d'autres gestionnaires d'environnement par exemple : Spack, Easybuild, ...

Utilisation de gestionnaires d'environnement logiciel

Fonctionnement général

- Initialisation du gestionnaire choisi (positionnement de variables d'environnement)
- Recherche et installation de logiciels dans le profil courant ou dans un profil spécifique
- On passe d'un profil à l'autre en activant un profil

Intérêts

- L'utilisateur est indépendant (pas d'installation système)
- La compilation des logiciels installés n'est pas faite manuellement
- L'utilisateur peut changer de version de logiciel utilisé à volonté
- L'utilisateur peut gérer plusieurs profils sauvegardés, isolés les uns des autres (projets)
- Facilite la recherche reproductible

Les gestionnaires d'environnement logiciel

L'intérêt pour la recherche reproductible

- L'environnement de base est (presque) vide
- On contrôle la version des logiciels installés et celle des dépendances
- Les anciens paquets restent disponibles même si il y a des nouvelles versions qui sont ajoutées
- A l'installation, on peut préciser la version exacte souhaitée
- Guix : on peut stocker dans un fichier l'environnement de travail dans lequel on est : nom et numéro de version de tous les logiciels installés (manifest)

Utilisation des gestionnaires d'environnement logiciel : Nix et Guix

Nix et Guix

- **Nix** et **Guix** remplacent *modules* présent avant sur les machines de Gricad
- Caractéristiques communes
  - Dépôt centralisé de paquets (store : /nix/store et /gnu/store)
  - Chaque paquet a plusieurs versions qui peuvent être installées et désinstallées indépendemment (avec les dépendances)
  - L'installation d'un logiciel est en fait une création de liens symboliques vers le dépôt centralisé
  - Permet à un développeur de packager et de distribuer un logiciel sans avoir à construire différents types de paquets pour les différents types de distributions.

Nix et Guix

Nix

- Sorti avant Guix : 2003
- Projet OpenSource
- NixOS

- Spécificités
  - Paquets écrits en langage fonctionnel nix (https://nixos.wiki/wiki/Nix_Expression_Language)
  - Documentation un peu dense
  - Logiciels non-libres
  - Importants nombre de paquets disponibles (+ 80 000 actuellement)
  - Disponible sous Linux et MacOS

Nix et Guix

Guix

- 2012
- Intégré au projet GNU (logiciel libre)
- GuixOS
- Initiative Guix-HPC (https://hpc.guix.info/)

- Spécificité
  - Paquets écrits en langage fonctionnel GNU/Guile
  - Ne propose que des logiciels libres
  - Dépôt est plus réduit en taille (nb paquets ≈21 000)
  - Disponible sous Linux

Nix et Guix

Utilisation concrète en quelques étapes

- Initialiser le gestionnaire choisi
- Chercher dans le dépôt le logiciel voulu
- Installer le paquet dans le profil de votre choix

Exemple d'initialisation du gestionnaire guix (cf. TP à suivre) :

```
#f-dahu%:source /applis/site/guix-start.sh
Default profile on
The following packages are currently installed in /home/cacar/.guix-profile/:
glibc-locales        2.33        out        /gnu/store/nrr24nvf6548if5wdpvxhlvjif3x9jjp-glibc-locales-2.33
nss-certs            3.71        out        /gnu/store/l0axwdizcc12n0kdicwlm7lcds7ym40d-nss-certs-3.71
```

Les gestionnaires Nix et Guix

Recherche d'un logiciel :

```
#f-dahu%:guix search gcc
name: gcc
version: 7.5.0
outputs: out lib debug
systems: x86_64-linux i686-linux
dependencies: gmp@6.2.1 isl@0.23 libelf@0.8.13 libstdc++-headers@10.3.0 mpc@1.2.1 mpfr@4.1.0 perl@5.34.0 texinfo@6.7 zlib@1.2.11
location: gnu/packages/gcc.scm:550:2
homepage: https://gcc.gnu.org/
license: GPL 3+
synopsis: GNU Compiler Collection
description: GCC is the GNU Compiler Collection.  It provides compiler front-ends for several languages, including C, C++, Objective-C,
+ Fortran, Ada, and Go.  It also includes runtime support libraries for these languages.
relevance: 27

name: gcc
version: 11.2.0
outputs: out lib debug
systems: x86_64-linux i686-linux
dependencies: gmp@6.2.1 isl@0.23 libelf@0.8.13 libstdc++-headers@10.3.0 mpc@1.2.1 mpfr@4.1.0 perl@5.34.0 texinfo@6.7 zlib@1.2.11
location: gnu/packages/gcc.scm:625:2
homepage: https://gcc.gnu.org/
license: GPL 3+
synopsis: GNU Compiler Collection
description: GCC is the GNU Compiler Collection.  It provides compiler front-ends for several languages, including C, C++, Objective-C,
+ Fortran, Ada, and Go.  It also includes runtime support libraries for these languages.
relevance: 27
```

Les gestionnaires Nix et Guix

Installation d'une version précise de gcc

```
#f-dahu%:guix install -p $GUIX_USER_PROFILE_DIR/ced2023 gcc@10.3.0
```

Activation du profil

```
#f-dahu%:refresh_guix ced2023
Activate profile  /var/guix/profiles/per-user/cacar/ced2023
The following packages are currently installed in /var/guix/profiles/per-user/cacar/ced2023:
gcc-toolchain        10.3.0        out        /gnu/store/bxh206gz379wkn8cvb2ghlkvpqgwfd2v-gcc-toolchain-10.3.0
```

Le gestionnaire d'environnements Conda

Introduction

Conda est un gestionnaire d'environnement incontournable, et on retrouve de nombreux exemples, tutoriels d'utilisation de logiciels qui se basent sur des environnements conda. Il a l'avantage d'avoir une courbe d'apprentissage assez directe. Il est important cependant de noter que la reproductibilité des environnements conda n'est pas toujours garantie.

- Charger l'environnement choisi : conda activate mon_env
- Chercher un logiciel : conda search nom_logiciel
- Installer le paquet : conda install nom_logiciel

Les cas d'usage :

A l'échelle du méso-centre le gestionnaire d'environnement conda est à utiliser lorsque les autres gestionnaires d'environnements n'ont pas pu répondre à vos besoins. Il est par exemple utilisé en IA pour bénéficier des frameworks incontournables comme Tensorflow, PyTorch, ou PyCUDA.

Singularity : travailler dans un conteneur

Introduction

A l'instar de docker, Singularity est un outils de gestion de conteneurs. Il permet donc d'instancier des images, de les créer à partir de "recettes" à la façon des dockerfiles. La conteneurisation peut être utilisée pour gérer son environnement de calcul dans le sens où elle est :

- **reproductible** : *les fichiers de définition* permettent de regénérer une image à l'identique.
- **indépendante du système hôte** : la conteneurisation permet d'isoler complètement le conteneur de son hôte.

Les cas d'usage :

Il existe de nombreux cas d'utilisation des containers en tant que gestionnaire d'environnement :

- le compilateur que l'on souhaite utilisé n'est pas disponible avec les autres gestionnaires d'environnements (ex: PGI)
- le logiciel sur lequel je travaille tourne déjà dans un environnement conteneurisé.

Singularity est le gestionnaire de conteneur utilisé à Gricad car il répond aux contraintes de sécurité, de support GPU, et supporte nativement les images docker.

Conclusions

Lequel choisir ?

- Connaître les spécificités pour en changer facilement
- Choisir celui le mieux adapté à votre application
- Les utiliser pour faire de la recherche reproductible

Le gestionnaire de "jobs" sur les machines de Gricad

Définition d'un gestionnaire de jobs

- Logiciel qui gère l'offre (les ressources disponibles) et la demande (les jobs en attente d'exécution)
- Ordonnanceur de job, batch scheduler ou gestionnaire de ressource

Rôle

- permet d'optimiser la gestion des ressources en classant les demandes / ressources (scoring)
- fournit une interface de soumission, avec une demande de ressource
- attribue le job sur la ressource demandée lorsque celle-ci devient disponible de façon exclusive

- On parle de gestion du cycle de vie d'un job :
  - Soumis au gestionnaire
  - En attente de ressources libres (Waiting)
  - En exécution (Running)
  - Terminé (Terminated)
- Exemples courants : PBS Pro, Maui, Torque, ...

Le gestionnaire de "jobs" sur les machines de Gricad : OAR

- OAR : gestionnaire de ressources pour grappe de PC ou cluster de calcul
- Développé originellement au laboratoire ID-IMAG (LIG) à Grenoble, comme un clone d'OpenPBS, sous licence OpenSource
- Utilisé par la plate-forme expérimentale nationale de grille de calcul Grid'5000, ainsi que par plusieurs autres sites universitaires
- Deux modes : interactif et batch
- *https://oar.imag.fr/*

Le gestionnaire de "jobs" sur les machines de Gricad : OAR

- Comment l'utiliser concrètement ? (cf. TP à suivre)
  - oarsub : soumission d'une demande de ressource

```
Exemple de commande de demande de ressources :
oarsub -l /nodes=1,core=2 --project airsea-modeling -I
```

  - oarstat : suivi de l'état du job

```
oarstat -u cacar (login)
Job id    S User     Duration   System message
--------- - -------- ---------- ------------------------------------------------
20447471  R cacar       0:41:28 R=34,W=2:0:0,J=I,P=airsea-modeling (Karma=0.001,quota
```

  - oardel : suppression d'un job

Autres utilitaires : chandler

```
cacar@f-dahu:chandler
```

Autres utilitaires : chandler

```
476 jobs, 3664 resources, 168 down, 3172 used

                Jobs       Jobs
User           running    waiting   Resources    Nodes
=============================================================
fernaang         0          7          0          0
valleemax        5          0          20         3
cooksa           0          1          0          0
zhuh             1          0          2          1
mikhaeln         40         0          40         7
ragvilla-ext     1          0          128        4
mangelo          2          3          256        8
sghojavand-ext   28         0          28         7
chaljube         1          45         64         2
mosbeuxc         1          0          32         1
bzizou           0          3          0          0
rouxphi          39         51         624        30
tomasetl         1          0          2          1
essingd          1          0          8          1
bracalem         1          0          128        4
nayakan          7          3          64         4
marteeti         3          0          384        12
madynar          9          1          76         5
lleshik          0          1          0          0
bazellep         0          1          0          0
provenzg         2          0          32         2
marir            15         0          240        12
atourette-ext    2          0          8          2
tesanv           1          0          20         1
vandenma         2          0          16         1
capognam         0          6          0          0
zandih           1          0          32         1
arjunans         5          0          192        7
moreaul          1          0          1          1
fachtonf         3          0          288        9
manceaua         3          0          64         3
montousj         1          0          8          1
zhangru          55         0          55         13
bonnevje         1          0          1          1
```

Autres utilitaires : recap.py

```
cacar@f-dahu:recap.py
```

```
================================================================================
    node    | cpumodel   |n_cpus n_cores| scratch1_type                | hasgpu |
            |            |   total_mem|                scratch2_type |        |
================================================================================
 bigfoot1 | Gold 6130 | 2 | 32 | 192 |system_ssd           | none               | YES    |
    [ + 1  more node(s) ]                                                       |
 bigfoot3 | Gold 6130 | 2 | 32 | 192 |system_ssd           | none               | YES    |
 bigfoot4 | Gold 5218R| 2 | 40 | 192 |system_ssd           | none               | YES    |
    [ + 1  more node(s) ]                                                       |
 bigfoot6 | Gold 5218R| 2 | 40 | 192 |system_ssd           | none               | YES    |
 dahu33   | Gold 6130 | 2 | 16 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu34   | Gold 6130 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
    [ + 37 more node(s) ]                                                       |
 dahu72   | Gold 6130 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu73   | Gold 6126 | 2 | 24 | 192 |system_ssd           | dedicated_hdd      | NO     |
    [ + 2  more node(s) ]                                                       |
 dahu76   | Gold 6126 | 2 | 24 | 192 |system_ssd           | dedicated_hdd      | NO     |
 dahu77   | Gold 6126 | 2 | 24 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
    [ + 3  more node(s) ]                                                       |
 dahu81   | Gold 6126 | 2 | 24 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu82   | Gold 6130 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
    [ + 23 more node(s) ]                                                       |
 dahu106  | Gold 6130 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu107  | Gold 6130 | 2 | 32 | 192 |dedicated_ssd        | none               | NO     |
 dahu108  | Gold 5218 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu109  | Gold 5218 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu110  | Gold 6244 | 2 | 16 | 192 |dedicated_ssd        | none               | NO     |
 dahu111  | Gold 6244 | 2 | 16 | 192 |dedicated_ssd        | none               | NO     |
 dahu112  | Gold 5218 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
    [ + 24 more node(s) ]                                                       |
 dahu137  | Gold 5218 | 2 | 32 | 192 |dedicated_ssd        | dedicated_hdd      | NO     |
 dahu138  | Gold 5218 | 2 | 32 | 192 |system_ssd           | dedicated_hdd      | NO     |
 dahu139  | Gold 5218 | 2 | 32 | 192 |system_ssd           | dedicated_hdd      | NO     |
 dahu140  | Gold 6244 | 2 | 16 | 192 |system_ssd           | dedicated_hdd      | NO     |
 dahu141  | Gold 6244 | 2 | 16 | 192 |system_ssd           | dedicated_hdd      | NO     |
 dahu-fat1| Gold 6244 | 2 | 16 |1500 |dedicated_raid0_ssd  | dedicated_raid5_hdd| NO     |
 dahu-fat2| Gold 6244 | 2 | 16 |1500 |dedicated_ssd        | dedicated_hdd      | NO     |
    [ + 2  more node(s) ]                                                       |
 dahu-visu| Silver 4216| 2 | 32 | 192 |system_hdd          | none               | NO     |
================================================================================
```

Utilisation des clusters de calcul - en résumé

Accès technique

- Accès: ssh
- Transfert de données : rsync, scp
- Git

Utilisation

0. Documentation: *https://gricad-doc.univ-grenoble-alpes.fr*
   SOS: *https://gricad-doc.univ-grenoble-alpes.fr/support/*
1. Passerelles de connexion: `Trinity, Rotule` $\longrightarrow$ *Aucune activité !*
2. Espaces de stockage: `scratch(bettik, silenus), home, tmp, Mantis` $\longrightarrow$ *Connaitre leurs usages!*
3. Environnements de calcul : `nix, guix, conda, singularity`
4. Soumission de jobs (interactif, batch): OAR

Prise en main des ressources de calcul

Nous avons créé un projet **trainings** dans le groupe **pole-calcul-formation/introduction-au-calcul-parallele**. Ce projet contient les énoncés et autres fichiers nécessaires aux exercices et tutoriels de ce module.
Pour y accéder, vous devez :

- avoir un compte sur la plateforme *https://gricad-gitlab.univ-grenoble-alpes.fr/*
- rejoindre le groupe pole-calcul-formation

Ensuite suivez les consignes indiquées sur la page du projet **trainings**.
Lien vers le projet : *https://gricad-gitlab.univ-grenoble-alpes.fr/pole-calcul-formation/introduction-au-calcul-parallele/trainings*.

- **Rappel des différentes étapes** :
  - Connexion SSH aux serveurs de calcul
  - Construction de l'environnemet logiciel de différentes façons
  - Soumission d job sur les clusters

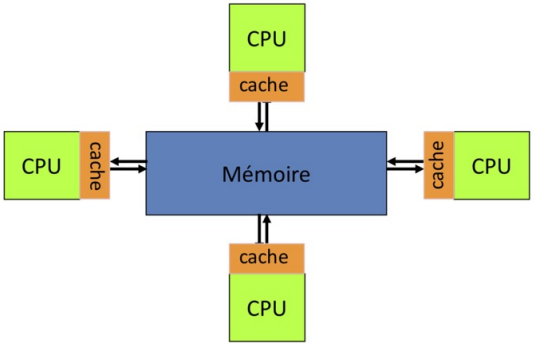
Calcul en mémoire partagée - Introduction à OpenMP - GPGPU

Céline Acary-Robert, Frédéric Audra, Glenn Cougoulat, Franck Pérignon

février-mars 2023

Systèmes à mémoire partagée, rappels

Un espace mémoire global, accessible simultanément par plusieurs "coeurs"/unité de calcul

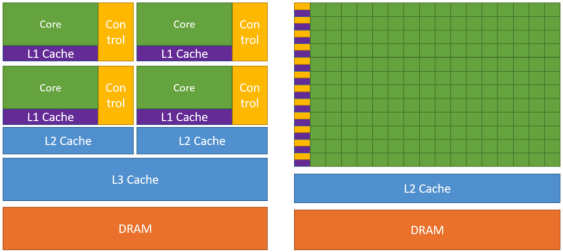


- Tous les CPUs accèdent à toute la mémoire globale, avec un même espace d'adressage.
- Chaque CPU est autonome et dispose de sa propre mémoire locale
- Toute modification dans la mémoire partagée est vue par tous les CPUs.

Système à mémoire partagée, rappels

- Une zone mémoire commune, accessible par tous les coeurs.
  - Accès aux données, échanges rapides.
  - Pas de gestion explicite de “communications” entre les processeurs.
  - Mécanisme de cohérence des caches.
- Ressources limitées : le code tourne sur un seul noeud de calcul.
- Mémoire maximum disponible = RAM du noeud.
- Réseau = bus, connectique entre coeurs et sockets.

CPU et GPU



- CPU : quelques dizaines de coeurs / GPU : des milliers de coeurs
- Les coeurs d'un GPU disposent de beaucoup moins de cache que sur un CPU
- La fréquence d'horloge est moins élevée sur un architecture GPU

Nous reviendrons en détails sur la comparaison GPU/CPU plus loin dans ce cours.

1 *OpenMP*
- Introduction et rappels
- Premiers pas avec OpenMP
- Régions parallèles
- Parallélisation des boucles
- Gestion du statut des variables
- Autres directives de partage du travail
- Tâches OpenMP
- Synchronisation
- Optimisations
- Conclusion

2 *GPGPU*

A propos d'OpenMP

OpenMP est une *interface de programmation* dédiée au calcul parallèle sur architecture à *mémoire partagée*.

- Specs définies par un consortium (*http://www.openmp.org/specifications/*).
- Membres permanents : AMD, ARM, Cavium, CRAY, Fujitsu, IBM, Intel, Micron, NEC, NVIDIA, Oracle, Red Hat, Texas instruments.
- Multi-plateformes (Linux, OSX, Windows).
- Disponible pour C, C++, Fortran.

A propos d'OpenMP

Définition d'un langage et d'une bibliothèque de fonctions qui vont vous permettre d'exploiter le parallélisme en mémoire partagée.

En pratique

- Plusieurs versions de la norme, chacune apportant de nouvelles fonctionnalités.
  - OpenMP 1.0 pour Fortran (1997) puis C/C++ (1998)
  - OpenMP 2.0 pour Fortran (2000) puis C/C++ (2002)
  - OpenMP 3.0 (2008)
  - OpenMP 4.0 (2013) : support accélérateurs, programmation SIMD, ...
  - OpenMP 5.0 (nov. 2018)
- Différentes implémentations, pour chaque version, selon le fournisseur du compilateur.
  OpenMP 4.5 pour GNU (>6.1), llvm, intel, ...
  Voir https://www.openmp.org/resources/openmp-compilers-tools/.

⇒ il faut toujours faire très attention au compilateur choisi et à la version disponible ! Il peut y avoir une différence entre la théorie, la norme et la réalité de ce qui est disponible/implémenté.

OpenMP

Une interface de programmation parallèle pour systèmes à mémoire partagée.
Principe : gestion de threads pouvant s'exécuter simultanément.

Thread/processus léger : fil d'exécution d'un ensemble d'instructions ($\approx$ programme) possédant sa propre **mémoire locale** (pile/stack, "threadprivate memory").

$\Rightarrow$ Deux types de variables : globales/partagées par tous les threads ou privées (dans la pile de chaque thread).

$\Rightarrow$ Les threads peuvent être organisés en team : ensemble de threads participant à l'exécution d'une région parallèle.

$\Rightarrow$ Pas besoin d'un cluster pour exploiter OpenMP.

Les principales fonctions d'OpenMP seront donc de :

- partager le travail entre les threads et gérer leur synchronisations,
- gérer les variables en mémoire, leur statut (privée/partagée).

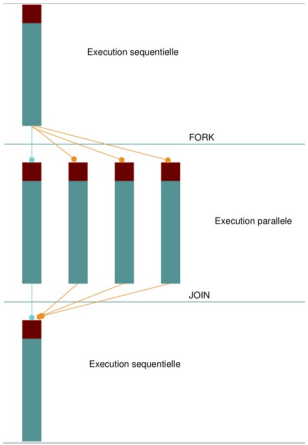
OpenMP

⚠ Ne pas confondre la couche logicielle (OpenMP, les threads, leurs mémoires associées) et la partie hardware (les coeurs, les caches, la RAM).

- La stack/mémoire “threadprivate” est une zone “réservée” au thread mais physiquement dans la RAM (mémoire commune aux différents coeurs de calcul).
- Les threads ont évidemment accès aux caches des coeurs de calcul.

Mode d'exécution "Fork-join"

OpenMP : une implémentation du modèle fork-join, alternance d'exécutions parallèles et séquentielles .



- Des "régions" séquentielles (code exécuté par un seul **thread maitre** de rang 0).
- Des régions parallèles : le code est exécuté sur plusieurs threads.
- Chaque thread est identifié par son rang .
- Chaque thread est affecté à un coeur physique. Remarque : l'ordre, les affectations dépendent de l'OS mais peuvent être controlés par l'utilisateur (affinité, numactl ...).

Résumons ...

Fonctionnement d'un programme OpenMP

- Un thread "maître" au démarrage. Apparition/disparition (fork-join) de threads "esclaves" au fil des régions séquentielles/parallèles.
- Chaque thread exécute sa séquence d'instructions, indépendamment des autres.
- Il a sa propre mémoire locale avec ses variables dites *privées*.
- Il accède à l'ensemble de la mémoire partagée et donc des *variables partagées*.
- L'ordre d'exécution des threads est défini par le système d'exploitation : *il faut s'assurer que l'accès aux variables partagées est correct : besoin éventuel de synchronisations*.
- ⚠ **Race condition** : mauvaise gestion de l'accès aux données pouvant entrainer des résultats variables d'une exécution à l'autre.

Elements de bibliographie

- La définition officielle de la norme :
  http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
  $\Rightarrow$ la référence a privilégier pour toutes les définitions, les prototypes de fonctions etc.
- Les cours de l'Idris : http://www.idris.fr/formations/openmp/

Structure générale d'un programme OpenMP

Un programme OpenMP est simplement un programme en C, C++ ou Fortran,

- avec des directives (insérées dans le code !) pour le compilateur, permettant la gestion des régions parallèles.

```
#pragma omp parallel ...
{// Code ...}
```

- utilisant les fonctions de la bibliothèque OpenMP qui permettent de contrôler l'environnement d'exécution

```
// Exemple : rang du thread courant
int rank = omp_get_thread_num();
```

- avec un pilotage par des variables d'environnement qui seront initialisées avant l'exécution du programme. Les variables d'environnement OpenMP (ICV dans la doc officielle) contrôleront l'exécution globale des régions parallèles.

Directives OpenMP

OpenMP utilise des directives (pragmas) pour le compilateur/pré-processeur afin de définir le comportement à l'exécution de certaines régions de code.

Syntaxe des directives :

```
sentinelle omp nom-directive [clause[ [,] clause] ... ]
```

- La "sentinelle" est la chaîne de caractères en début de ligne qui annonce une instruction OpenMP : !$omp en fortran / #pragma omp en C/C++.
- Les clauses permettent de définir les options de la directive utilisée.

Exemple

C/C++

```
#pragma omp parallel for
```

Fortran

```
!$omp parallel do
! alternatives : c$omp, *$omp
```

On notera dans la suite du cours DIRECTIVE(code fortran) / DIRECTIVE(code C/C++) CLAUSE

OpenMP, runtime library

Une bibliothèque de fonctions C ou fortran

- Gestion des threads
- Synchronisation
- Timers ...

Comment l'utiliser ? Il suffit (et il faut !)

d'inclure un fichier omp.h en C/C++

```
#include <omp.h>
```

ou d'appeler le module OMP_LIB en fortran

```
!$ use OMP_LIB
```

L'utilisation de la librairie à l'exécution et l'interprétation des directives sera déterminée par une option ajoutée à la ligne de compilation/édition de lien, comme nous verrons plus loin.

OpenMP, runtime library, quelques exemples

Gestion des threads

```
omp_set_num_threads()  // fixe le nombre de threads à l'exécution
omp_get_num_threads()  // retourne le nombre de threads impliqués dans la ``team''
omp_get_max_threads()  // le nombre max de threads pouvant être utilisés
                       // par le programme
omp_get_thread_num()   // le numero du thread appelant dans le groupe (de 0 à n-1)
omp_get_num_procs()    // renvoie le nombre de coeurs disponibles pour le programme
omp_in_parallel()      //  booléen qui indique si le thread appelant
                       //  est dans une région parallèle
```

Calcul du temps d'exécution

```
start = omp_get_wtime() // renvoie le temps écoulé (secondes) depuis une date arbitraire fixe
// Calculs ...
elapsed = omp_get_wtime() - start  // temps écoulé, mesure pour le thread courant
```

De nombreuses autres fonctions sont disponibles. Voir *http://www.openmp.org/specifications/* pour une liste exhaustive et les détails de l'API.

Compilation conditionnelle

OpenMP est conçu de manière à permettre, pour un même code, une compilation et une exécution correcte en parallèle ou en séquentiel (i.e. si OpenMP n'est pas disponible ou pas activé).

- Les directives sont considérées par le compilateur comme des lignes de commentaires si OpenMP n'est pas "activé" à la compilation.
- Pour la compilation conditionnelle d'une fonction (user-defined ou de la bibliothèque OpenMP), d'une région, il faut utiliser :

la sentinelle !$ en fortran

```
!$ call do_something_parallel()
```

la macro *_OPENMP* en C/C++

```
#ifdef _OPENMP
  do_something_parallel();
#endif
```

Ces lignes seront ignorées si OpenMP n'est pas activé à la compilation !
👍Utilisez ces outils pour écrire du code qui fonctionne avec ou sans OpenMP.

Variables d'environnement ou Internal Control Variables (ICV)

Les variables d'environnement ou ICV permettent de fixer à l'exécution un certain nombre de paramètres.
Elles sont déclarées/initialisées avant l'exécution :

```
# fixe le nombre de threads impliqués
export OMP_NUM_THREADS=4
# type de répartition des itérations de boucle
export OMP_SCHEDULE="dynamic"
```

Pour une liste complète des directives, fonctions et variables d'environnement, voir les aides mémoire OpenMP :

- en C/C++ *http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf*
- en Fortran *http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-F-web.pdf*

Compilation et exécution

Pour compiler/linker : une option à ajouter, qui va bien entendu dépendre du compilateur (*https://www.openmp.org/resources/openmp-compilers-tools/*).

- **Compilation/link** : -fopenmp avec gnu, -qopenmp avec intel

```
gfortran -fopenmp prog.f90  -o exemple
gcc -fopenmp prog.c -o exemple
```

- **Exécution** : choix du nombre de threads à utiliser puis exécution.

```
export OMP_NUM_THREADS=4
./exemple
```

La valeur par défaut de OMP_NUM_THREADS va dépendre de la plate-forme (a priori 1 ?).
👍Penser à toujours renseigner cette variable explicitement .

- Vérification de la **version** d'openmp

```
echo |cpp -fopenmp -dM |grep -i open # pour gnu
icc -qopenmp -E -dM - < /dev/null |grep -i openmp # pour intel
```

Résumons ...

Caractéristiques d'un programme OpenMP

- Des directives insérées dans un code classique
- La syntaxe des directives dépend du langage
- Des variables d'environnement (ICV) qui conditionnent l'exécution
- Une bibliothèque de fonctions
- La compilation est conditionnelle : le code peut s'exécuter en séquentiel
- Une option est donnée au compilateur pour prendre en compte OpenMP

Premiers exemples de partage du travail

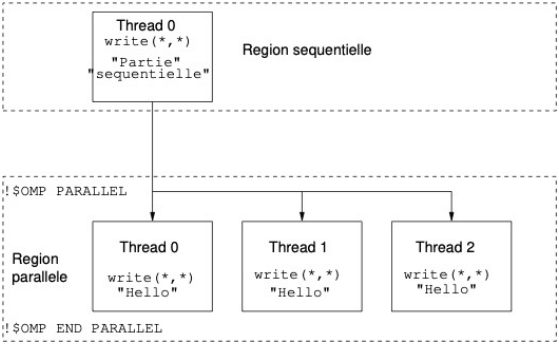
Le rôle principal d'OpenMP est de partager une charge de travail, décrite dans un programme, entre des threads. Deux approches principales pour gérer ce partage :

- La distribution de tâches autonomes sur plusieurs threads ($\approx$ coarse-grain, task parallelism).
- La parallélisation de boucles, ($\approx$ fine-grain, data parallelism) distribution des itérations entre les threads. On découpe et distribue une tâche (la boucle) sur plusieurs threads.

Construction d'une région parallèle

Principe : création d'une "équipe" de threads qui vont tous exécuter le bloc de code de la région (souvenez vous du modèle fork-join), via la directive PARALLEL

```
program parallel
implicit none
write(*,*) "Partie sequentielle"

!$OMP PARALLEL
  write(*,*) "Hello"
!$OMP END PARALLEL

end program parallel
```



- Création implicite d'une tâche par thread.
- Dans la région parallèle, tous les threads exécutent la même portion de code.
- Pas d'ordre a priori.
- Synchronisation implicite en fin de région.

Construction d'une région parallèle

Dans l'exemple précédent, la fin de la région parallèle dans le code fortran est définie par la ligne

```
!$OMP END PARALLEL
```

On retrouvera cette construction (OMP DIRECTIVE ... OMP END DIRECTIVE) pour la plupart des directives.
En C/C++, pas de "end", on utilisera les { } pour délimiter les portées des directives.
Par exemple

```
#pragma omp parallel
{
   ...
}
```

Construction d'une région parallèle, exemple

```
program parallel
!$ use OMP_LIB
implicit none
real :: a
logical :: p
a = 92290; p=.false.
!$OMP PARALLEL
!$ p = OMP_IN_PARALLEL()
print *,"A vaut : ",a
!$OMP END PARALLEL
print*,"Parallele ?:", p
end program parallel
```

Compilation/link/exécution avec ifort

```
> ifort -qopenmp prog.f90
> export OMP_NUM_THREADS=3
> ./a.out
A vaut : 92290
A vaut : 92290
A vaut : 92290
Parallele ? : T
```

Construction d'une région parallèle

Chaque thread recoit un numéro par lequel il sera identifié tout au long de l'exécution du programme. Ce numéro est appelé "rang", obtenu via la fonction :

```
int rang = omp_get_thread_num();
```

Cela permet (entre autres) d'inclure des tests sur ce rang et de confier ainsi des portions de code à un thread en particulier :

```
program parallel
!$ use OMP_LIB
implicit none
real :: a
logical :: p
a = 92290; p=.false.
!$OMP PARALLEL private (rang)
!$ p = OMP_IN_PARALLEL()
rang=omp_get_thread_num()
if (rang==3) print *,"A vaut : ",a
!$OMP END PARALLEL
print*,"Parallele ?:", p
```

Imbrications de régions parallèles

OpenMP permet l'inclusion (nesting) d'une deuxième zone parallèle dans une zone parallèle existante.
Si on définit simplement une nouvelle région dans une région existante :

```
#pragma omp parallel
{
 int rang = omp_get_thread_num();
 int nb = omp_get_num_threads();
 printf("main :: thread num %d sur %d\n", rang, nb);
 #pragma omp parallel
 { // sous-region'work'
 int rang = omp_get_thread_num();
 int nb = omp_get_num_threads();
 printf("work :: thread num %d sur %d\n", rang, nb);
 }
}
```

```
 > g++ -fopen main.cpp
 > ./a.out
main :: thread num 0 sur 2
work :: thread num 0 sur 1
main :: thread num 1 sur 2
work :: thread num 0 sur 1
```

Le "nesting" ne fonctionne pas !

Imbrications de régions parallèles

Pour pouvoir imbriquer des régions parallèles :

- il faut positionner la variable d'environnement OMP_NESTED à TRUE avant l'exécution,
- OU faire appel à la fonction omp_set_nested() dans le programme.

```
#pragma omp parallel
{
 int rang = omp_get_thread_num();
 int nb = omp_get_num_threads();
 printf("main :: thread num %d sur %d\n", rang, nb);
 #pragma omp parallel
 { // sous-region'work'
 int rang = omp_get_thread_num();
 int nb = omp_get_num_threads();
 printf("work :: thread num %d sur %d\n", rang, nb);
 }
}
```

```
> export OMP_NESTED=TRUE
> g++ -fopen main.cpp
> ./a.out
main :: thread num 0 sur 2
main :: thread num 1 sur 2
work :: thread num 0 sur 2
work :: thread num 1 sur 2
work :: thread num 0 sur 2
work :: thread num 1 sur 2
```

Chaque thread impliqué dans la première zone parallèle, génère une nouvelle équipe de threads à l'entrée de la deuxième zone.

Résumons ...

Création de zones parallèles

- On insère les directives de début et fin de zone parallèle : directive PARALLEL / parallel
- Chacun des threads exécute le code de la zone parallèle
- L'imbrication de zones parallèles nécessite une action supplémentaire (variable d'environnement ou appel à la fonction omp_set_nested()

Exercices : prise en main du cluster et premier programme OpenMP

Rendez-vous dans le projet habituel pour faire la partie **OpenMP - Prise en main l'environnement de travail - Ecriture et exécution d'un premier programme parallèle**.

Parallélisation de boucles

Nous avons vu comment créer une région parallèle et faire exécuter une tâche par une équipe de threads via la directive parallel .

Nous allons maintenant nous intéresser à la parallélisation de boucles : l'idée étant de découper une ou plusieurs boucles en paquets d'itérations et de distribuer ces paquets sur les threads pour une exécution simultanée .

Remarque : la parallélisation de boucles est en général la première étape, simple et facile à mettre en oeuvre, de parallélisation d'un programme.

Parallélisation de boucles

Cela consiste à :

- Définir une région parallèle contenant la ou les boucles, via la directive parallel que nous avons déja vue,
- Puis à répartir les itérations de la boucle entre les threads en utilisant la directive DO / for.
- Il est possible d'utiliser directement une seule directive PARALLEL DO / parallel for, contraction des deux précédentes.

Parallélisation de boucle

Choix du nombre de threads

Programme Séquentiel

```
void main()
{

 for(int i=0;i<1000;i++)
 do_something(i);

 }
```

Programme Parallèle

```
void main()
{
  omp_set_num_threads(4);
  #pragma omp parallel for
  for(int i=0;i<1000;i++)
    do_something(i);
}
```

Répartition des 1000 itérations sur 4 threads

Très simple : une seule ligne de code à ajouter !

Pas de modification de la structure du programme.

Remarque : à titre d'exemple, le nombre de threads est ici fixé explicitement, ce qui est en général à proscrire pour des questions de portabilité, on préfèrera initialiser la variable d'environnement OMP_NUM_THREADS.

Parallélisation de boucle, quelques remarques

- Seule la boucle suivant l'instruction parallel do / parallel for est distribuée sur les threads ("outer loop").
- On verra qu'il est possible d'imbriquer les directives do / for dans une région parallèle.
- Toutes les opérations réalisées lors d'une itérations doivent être indépendantes les unes des autres.
- Les variables utilisées pour définir les bornes de la boucle (indices) ne doivent pas être modifiées par celle-ci.
- Les itérations attribuées à un thread (appelé chunk) sont réalisées dans l'ordre sur le thread concerné mais on ne connait pas a priori l'ordre d'exécution des threads !
- Le mode de répartition par défaut des itérations sur les threads dépend de l'implémentation (voir schedule ci-après)

Parallélisation de boucle, quelques remarques

- Synchronisation implicite de tous les threads en sortie de boucle.
- Pour empêcher cette synchronisation, il faut utiliser la clause NOWAIT.
  - ⇒ meilleures performances eventuelles,
  - ⇒ au prix d'une gestion explicite, par le programmeur, des synchronisations (e.g. via la directive BARRIER).

```
#pragma omp parallel
{
  #pragma omp for nowait
  for(...)
  // ...
  // pas de synchro en fin de boucle
  ...
  // Barrière de synchronisation des threads
  #pragma omp barrier
}
```

Répartition des itérations (schedule)

Schedule : un outil pour contrôler la répartition des itérations sur les threads.

- Combien d'itérations par thread (on parlera de "chunk size") ?
- Comment ("kind") ces chunks sont-ils distribués (dans quel ordre, quand etc) ?
- schedule (kind, chunk-size)

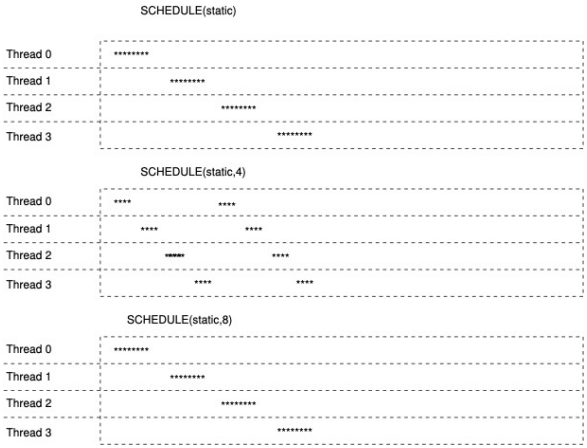
Exemple :

```
!$omp parallel do schedule(STATIC,4)
do i=1,1000
!! ...
end do
!$omp end parallel do
```

Modes/kind possibles : STATIC, DYNAMIC, GUIDED, RUNTIME, AUTO

Répartition des itérations : mode STATIC

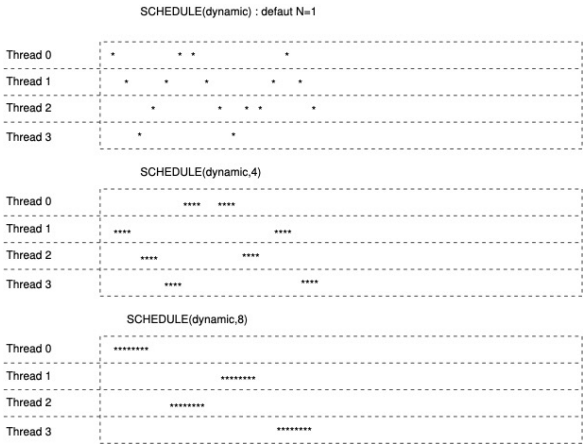
La clause SCHEDULE(STATIC,N) permet de diviser les itérations en groupe de taille ***N (chunk-size)***.



- **Répartition circulaire (round-robin)** entre les threads, par ordre croissant de rang des threads.
- Ordonnancement à la compilation, gestion par le programmeur.
- Si N n'est pas précisé, découpage en chunk plus ou moins égaux.

Répartition des itérations : mode DYNAMIC

La clause SCHEDULE(DYNAMIC, N) divise les itérations en groupe de taille N sans a priori sur l'ordre de répartition entre les threads.



- ⚠ ordonnancement au runtime : chaque thread effectue N itérations puis demande une nouvelle affectation d'itérations jusqu'à ce que le calcul soit terminé. Coûteux, attention à l'overhead !
- chunk-size, défaut $= 1$.

Répartition des itérations : compléments

- SCHEDULE(GUIDED, N) : fonctionnement très proche de celui obtenu avec la clause DYNAMIC.
  La taille des *chunks* change à chaque affectation et est proprotionnelle au nombre d'itérations restantes divisé par le nombre de threads. N (defaut = 1) indique la taille minimum des chunks.
- SCHEDULE(AUTO) : laisse le compilateur ou l'exécution décider de la répartition.
- SCHEDULE(RUNTIME) : la répartition est décidée à l'exécution via la variable d'environnement *OMP_SCHEDULE* (👍: portabilité du code).

```
export OMP_CHEDULE="GUIDED,4"
```

Répartition des itérations : compléments

Le choix du mode de répartition doit respecter quelques principes :

- Pour optimiser les performances, la charge par thread ne doit pas être trop faible (load balancing).
- Attention au risque de *false-sharing*.
  Si deux threads traitent des portions de tableaux trop proches en mémoire, ils chargent dans leur cache local la même zone de mémoire partagée.
  Dans ce cas il y a un risque de perte de cohérence entre le cache et la donnée en mémoire.
  A l'exécution, cette perte de cohérence est corrigée mais *cela induit une perte de performance*.

Traitement des boucles imbriquées

- La directive DO permet de paralléliser uniquement la boucle qui suit la directive.
- Quid des boucles imbriquées ?
- Deux possibilités :
  - Imbriquer les régions parallèles et les directives DO / for
  - Utiliser la clause COLLAPSE

Imbrications de boucles

```
#pragma omp parallel for
for (int j=0;j<jmax;j++)
{
  for(int i=0;i<imax;i++){
    printf("i,j, rang : %d,%d,%d\n", i,j, omp_get_thread_num());
  }
}
```

Par défaut, la directive DO / for permet de paralléliser uniquement la boucle qui suit la directive : dans cet exemple, seule la boucle externe est parallélisée.

Imbrications de boucles

Il est possible d'imbriquer les zones parallèles pour répartir les itérations des deux boucles. Pour paralléliser une deuxième boucle interne à la première *on crée une deuxième zone parallèle* (N'oubliez pas l'ICV OMP_NESTED !).
Notre exemple devient :

```
#pragma omp parallel for
for (int j=0;j<jmax;j++)
{
  #pragma omp parallel for
  for(int i=0;i<imax;i++){
    printf("i,j, rang : %d,%d,%d\n", i,j, omp_get_thread_num());
  }
}
```

Parallélisation de boucle : clause COLLAPSE

Il existe une autre solution, plus optimale, pour imbriquer des boucles : la clause collapse(N), N étant le nombre de boucles à paralléliser.

- Il s'agit de fusionner les boucles imbriquées et de répartir ensuite les itérations pour *potentiellement améliorer les performances*

```
#pragma omp parallel for collapse(2)
for (int j=0;j<jmax;j++)
  for(int i=0;i<imax;i++){
    printf("i,j, rang : %d,%d,%d\n", i,j, omp_get_thread_num());
  }
```

Résumons ...

Parallélisation de boucles

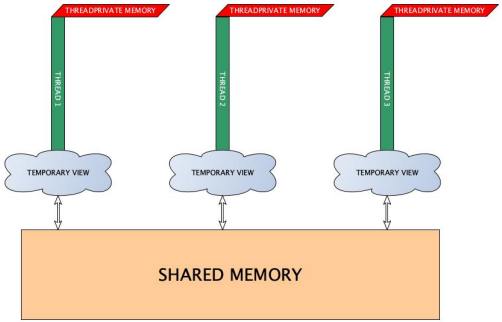
- On insère les directives de parallélisation de boucles : PARALLEL DO / parallel for
- La répartition des itérations est paramétrable par l'utilisateur
- L'imbrication de boucles se fait soit par l'imbrication de régions parallèles ou par la clause COLLAPSE plus optimale

Statut des variables - Gestion de la mémoire en OpenMP

Chaque thread dispose de deux espaces mémoires distinctes pour stocker les variables qu'il utilise :

- une **mémoire privée**, (pile/stack/threadprivate memory) inaccessible pour les autres threads,
- une zone **mémoire commune (shared)**, partagée avec tous les autres threads.

Il existe également une zone intermédiaire nommée **temporary view** qui permet au thread de stocker temporairement certaines variables partagées, dans un but d'optimisation, afin d'éviter de multiplier les accès à la mémoire commune. Cette zone peut appartenir aux registres, caches ou autres du coeurs hébergeant le thread.



Statut des variables - Gestion de la mémoire en OpenMP

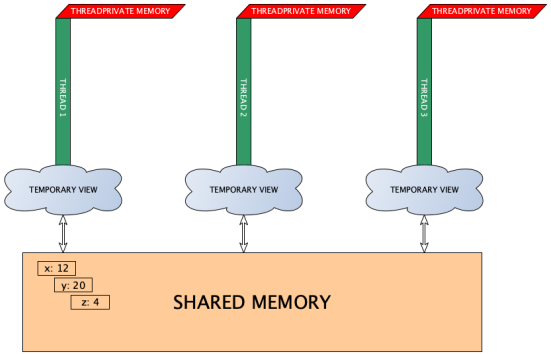
Dans un programme Openmp, il est possible (et nécessaire !) de fixer le statut d'une variable pour déterminer son appartenance à une zone ou une autre.
Ce statut (shared ou private) est un paramètre fondamental dans un programme OpenMP. Il impactera les résultats (valeur des variables, race condition) les performances (false sharing ...)

- Par défaut les variables sont allouées dans la zone partagée par les threads.
- Une variable avec un statut privée sera allouée dans la pile du thread.

Le statut d'une variable est piloté via les clauses shared, private et les clauses similaires pour une gestion plus fine du contenu de la variable : initialisation, transfert shared/private etc.

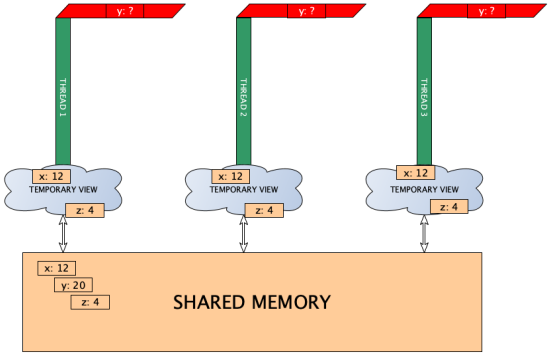
Statut des variables

```
double x = 12., y = 20., z = 4.;
#pragma omp parallel shared(x) private(y)
{
  # z : shared par défaut.
  # Modifications de x, y, z
  x = 3;
  y = omp_get_thread_num();
  z = 17;
}
```



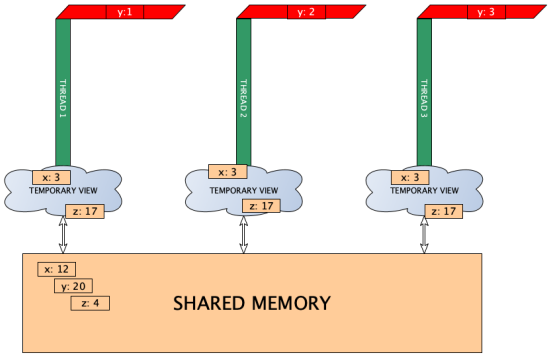
Statut des variables

```
double x = 12., y = 20., z = 4.;
#pragma omp parallel shared(x) private(y)
{
  # z : shared par défaut.
  # Modifications de x, y, z
  x = 3;
  y = omp_get_thread_num();
  z = 17;
}
```



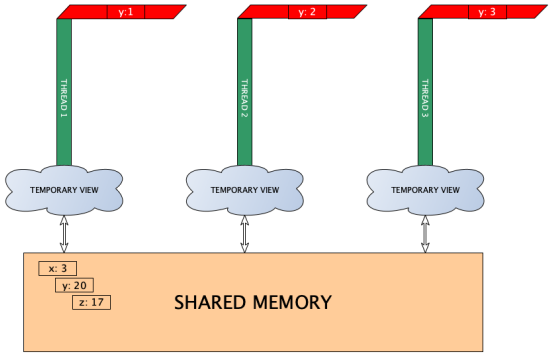
Statut des variables

```
double x = 12., y = 20., z = 4.;
#pragma omp parallel shared(x) private(y)
{
  # z : shared par défaut.
  # Modifications de x, y, z
  x = 3;
  y = omp_get_thread_num();
  z = 17;
}
```



Statut des variables

```
double x = 12., y = 20., z = 4.;
#pragma omp parallel shared(x) private(y)
{
  # z : shared par défaut.
  # Modifications de x, y, z
  x = 3;
  y = omp_get_thread_num();
  z = 17;
}
```



⚠ Le même nom de variable dans le programme peut désigner deux zones mémoires différentes.
⚠ Une variable privée n'est pas initialisée (contenu indéterminé !) à l'entrée de la zone parallèle. Voir y dans l'exemple ci-dessus.

Statut des variables : clause PRIVATE, exemple fortran

```
integer :: rang, a ! variables 'shared'
a = 92000

!$OMP PARALLEL PRIVATE(rang,a)
! a et rang sont privées, leur valeur est indéterminée.
!$ rang = OMP_GET_THREAD_NUM()
a = a + rang + 290
print *,"Rang : ",rang,"; A vaut : ",a
!$OMP END PARALLEL

! a et rang désignent à nouveau une zone
! en mémoire partagée
print*,"Hors region, A vaut :",a
```

```
Hors region, A vaut : 92000
Rang : 1 ; A vaut : 291
Rang : 2 ; A vaut : 292
Rang : 0 ; A vaut : 290
Rang : 3 ; A vaut : 293
Hors region, A vaut : 92000
```

Statut des variables : clause PRIVATE

⚠ en C++ à la déclaration des variables au fil de l'eau.
Les deux écritures ci-dessous sont possibles :

```
double var = 38;
#pragma omp parallel private(var)
{
  // var est privée, sa valeur
  // est indéterminée
}
```

```
double var = 38;
#omp pragma omp parallel
{
  // déclaration d'une var locale
  // à la zone parallèle
  double var = 4;
}
printf("Var =  %f\n", var);
// --> affiche 38
```

Statut des variables : clause firstprivate

firstprivate : permet d'initialiser la variable privée à la dernière valeur qu'elle avait avant l'entrée dans la région parallèle.

```
real :: a
a = 92000

!$OMP PARALLEL firstprivate(a)

! a initialisée avec sa valeur en zone 'shared'

a = a + 290 + rang
print *,"Rang : ",rang,"; A vaut : ",a
!$OMP END PARALLEL

! Retour à la valeur initiale (hors zone privée)
print*,"Hors region, A vaut :",a
```

```
Rang : 1 ; A vaut : 92291
Rang : 2 ; A vaut : 92292
Rang : 0 ; A vaut : 92290
Rang : 3 ; A vaut : 92293
Hors region, A vaut : 92000
```

Statut des variables : clause lastprivate

- lastprivate : la variable est privée et son contenu est conservé en sortie de bloc.
- Où ? Uniquement Directives de type boucles (e.g. DO / for) et SECTIONS.
- La valeur conservée est celle de la dernière itération ou de la dernière section.

```
real :: a
a = 92000
!$OMP PARALLEL
!$OMP DO lastprivate(a)
! a privée, de valeur indéterminée
do i=1,1000
   a = i
end do
print *,"A vaut : ",a
!$OMP END PARALLEL
! a contient la dernière valeur calculée en zone privée
```

```
A vaut :     1000.00000
A vaut :     1000.00000
A vaut :     1000.00000
A vaut :     1000.00000
```

Statut des variables : clause DEFAULT

DEFAULT(arg) : impose un comportement par défaut pour le statut des variables, s'il n'est pas explicite.

arg = (NONE|SHARED|PRIVATE|firstprivate) / (none|shared)

```
#pragma omp parallel default(none) shared(var, n) private(m)
{
  // ...
}
```

👍La clause DEFAULT(none) oblige le programmeur à être explicite pour définir le statut des variables.

Statut des variables : allocation dynamique

L'allocation dynamique de variable peut être effectuée au sein d'une région parallèle.

- Cas d'une variable privée : elle reste locale à chaque thread.
- Cas d'une variable partagée : un seul thread doit se charger de l'allocation, mais il vaut mieux que tous les threads participent à l'initialisation.

```
// initialisation des données
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }

#pragma omp parallel for private(j)
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }
```

```
#pragma omp parallel for private(j)
// initialisation des données
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }

#pragma omp parallel for private(j)
for(i=0; i<N; i++)
    for(j=0; j<M; j++) { … }
```

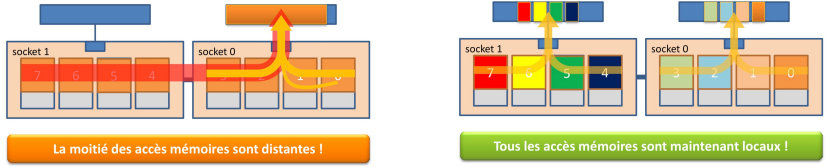


Figure: source : séminaire MaiMoSiNE L.Saugé.

Statut des variables : sous-programme et transmission par arguments

- L'influence (ou la portée) d'une région parallèle s'étend aussi bien au code contenu lexicalement dans cette région (étendue statique), qu'au code des sous-programmes appelés. L'union des deux représente *l'étendue dynamique*.
- Dans un *sous-programme* appelé dans une région parallèle, les variables locales et tableaux automatiques sont *implicitement privés* à chaque tâche.
- Dans une procédure, toutes les variables transmises par argument héritent du statut défini dans l'étendue lexicale (statique) de la région.

Statut des variables : sous-programme et transmission par arguments

```
void fonction (int a, int &b){
  b = a + omp_get_thread_num();
}
int main(){
  int a = 5, b;
  omp_set_num_threads(6);
  #pragma omp parallel default(none) shared(a) private(b)
  {
   fonction(a, &b);
   printf("%d-a=%d,b=%d\n",omp_get_thread_num(),a,b);
 }
 // contenu de b indéterminé
 // en dehors de la zone parallèle !
  return EXIT_SUCCESS;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=4
> a.out

3 - a=5, b=8
4 - a=5, b=9
0 - a=5, b=5
1 - a=5, b=6
2 - a=5, b=7
5 - a=5, b=10
```

Réductions

Réduction : Copie privée ET initialisation d'une liste de variables puis application d'une opération à la fin de la zone parallèle, e.g.

```
#pragma omp parallel reduction(+:var1, var2) // opération '+', appliquée à var1 et var2
{
  // var est initialisée localement à 0
  change(var1, var2) // modifications locales
}
// en sortie, var1 = somme de sa valeur sur chaque thread. Idem pour var2.
```

L'opération peut être de plusieurs types :

- arithmétique : +, -, *, /
- logique : AND, OR, EQV, NEQV
- fonction intrinsèque : MAX, MIN, etc

La valeur d'initialisation dépend de l'opérateur. Voir par exemple *https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf*.

Exemple: réduction dans une boucle

```
integer, parameter :: n=5
integer :: i, s=0, p=1, r=1
!$OMP PARALLEL
!$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
do i = 1, n
  s = s + 1
  p = p * 2
  r = r * 3
  print *,'rang',omp_get_thread_num(),'iter =',i
end do
!$OMP END DO
print *,'rang', omp_get_thread_num(),
        's =',s, '; p =',p, '; r =',r
!$OMP END PARALLEL
print *,"s =",s, "; p =",p, "; r =",r
```

```
rang 0   iter 1
rang 0   iter 2
rang 2   iter 4
rang 3   iter 5
rang 1   iter 3

rang 0  s = 5 ; p = 32 ; r = 243
rang 3  s = 5 ; p = 32 ; r = 243
rang 2  s = 5 ; p = 32 ; r = 243
rang 1  s = 5 ; p = 32 ; r = 243

s = 5 ; p = 32 ; r = 243
```

Résumons ...

Status des variables

- Chaque variable a un statut propre
- Le statut est défini en début de zone parallèle : variable privée private ou partagée shared
- Un statut par défaut peut être positionné en début de zone parallèle
- L'initialisation dépend du statut mais peut être modifié par des clauses spécifiques : firstprivate, lastprivate
- Une opération de réduction peut être effectuée en sortie de zone parallèle

Exercices

Rendez-vous dans le projet habituel pour faire les parties

- OpenMP - Statut des variables
- OpenMP - Parallélisation de boucles

Partage du travail

Nous avons vu jusqu'à présent une seule méthode de partage du travail, do / for. Il en existe d'autres que nous allons passer en revue :

- Sections
- Single / Master
- Workshare (Fortran only !)
- Tasks (voir section suivante)

Chacune de ces construction :

- doit faire partie d'une région parallèle (et est donc associé à une équipe de threads),
- doit être "vue " par tous les threads (ou aucun) de la région,

Les synchronisations dépendent de chaque directive.

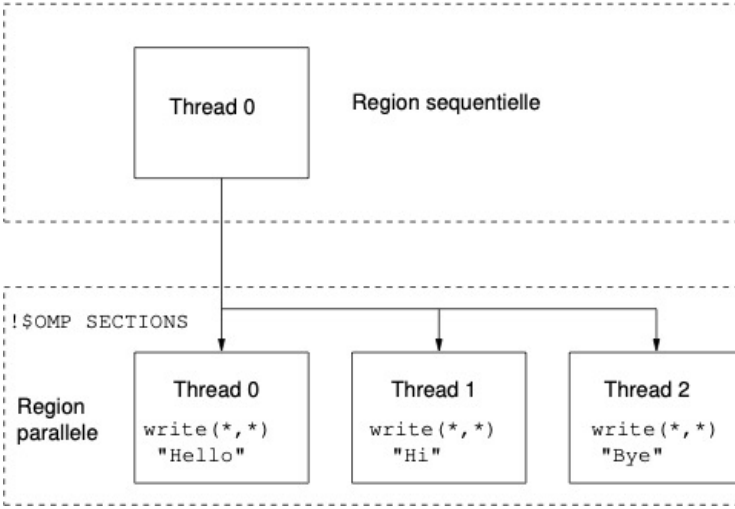
Sections parallèles : directives SECTIONS/SECTION

SECTIONS : attribution de portions de codes différerentes à chaque thread de la région parallèle.
SECTION : une portion de code exécutée par un seul thread

- Permet d'attribuer des portions de codes totalement indépendantes à différents threads.
- Barrière de synchronisation implicite en fin de construction SECTIONS.
- La clause NOWAIT est admise pour SECTIONS.

Sections parallèles : exemple

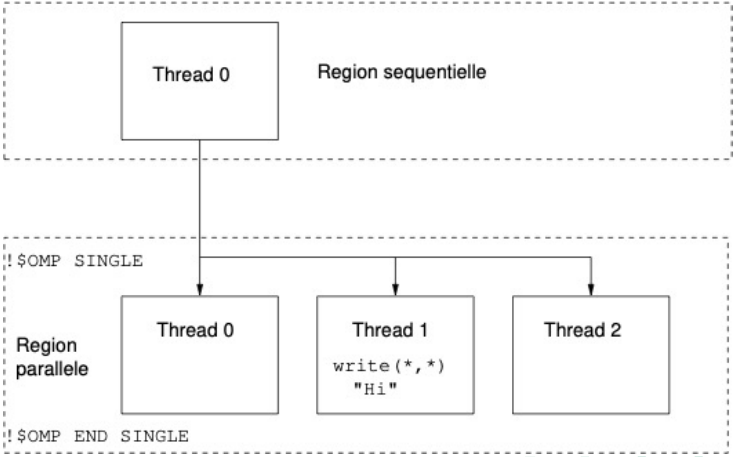
```
// Définition d'un ensemble
// de sections
#pragma omp parallel sections
{
  // Définition/distribution
  // des différentes sections
  #pragma omp section
  work1(); // printf("Hello");
  #pragma omp section
  Work2(); // printf("Hi");
  #pragma omp section
  Work3(); // printf("Bye");
}
```



Exécution exclusive : SINGLE

Exécution d'un bloc de code par un (et un seul) thread + synchronisation.

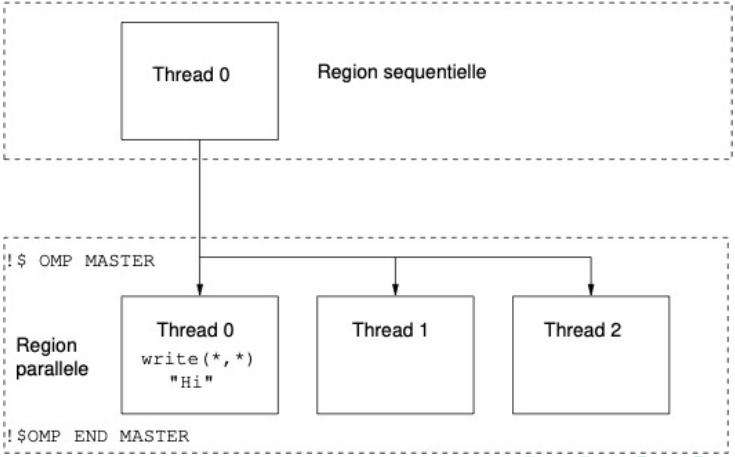
```
#pragma omp parallel
{
...
#pragma omp SINGLE
{
 printf("Hi");
}
// synchro implicite
// à la fin du block!
// possibilité de nowait.
 ...
}
```



Exécution exclusive : MASTER

Exécution d'un bloc de code par le thread master SANS synchronisation.

```
#pragma omp parallel
{
...
#pragma omp master
{
 printf("Hi");
}
// pas de synchro,
// les autres threads
// ignorent le block
 ...
}
```



Compléments : directive WORKSHARE

⚠ Fortran only !

- Rôle : permet la répartition du travail dans certains cas liés à l'utilisation du Fortran 95
- Ne s'utilise qu'au sein d'une région parallèle
  - Fonctions intrinsèques portant sur des tableaux (exemples: MATMUL, DOT_PRODUCT, SUM, COUNT, etc ...)
  - Instructions de type FORALL et WHERE
- N'admet que la clause NOWAIT en fin de construction

Directive WORKSHARE, exemple

```
integer, parameter :: m=4097, n=513
 integer :: i, j
real, dimension(m,n) :: a, b
call random_number(b)
a(:,:) = 1.
!$OMP PARALLEL
!$OMP DO
 do j=1,n
   do i=1,m
      b(i,j) = b(i,j) - 0.5
   end do
 end do
!$OMP END DO
!$OMP WORKSHARE
WHERE(b(:,:) >= 0.) a(:,:)=sqrt(b(:,:))
! $OMP END WORKSHARE NOWAIT
!$OMP END PARALLEL
```

Compléments : choix du nombre de threads

- Variable d'environnement OMP_NUM_THREADS=N
- OMP_NUM_THREADS=N1,N2 pour chaque niveau de nesting
- Fonction omp_set_num_threads(N)
- La directive PARALLEL admet la clause NUM_THREADS

```
#pragma omp parallel num_threads(3)
{...} // Une region avec 3 threads
```

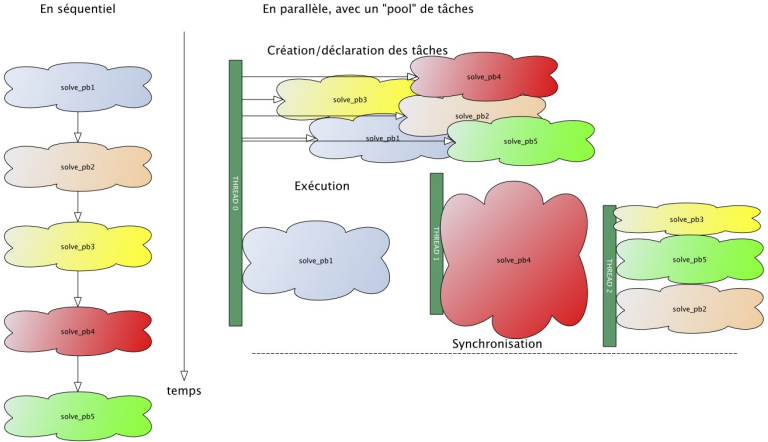
Résumons ...

Autres directives de partage du travail

- SECTIONS, SECTION : permet d'attribuer des portions de codes totalement indépendantes à différents threads,
- SINGLE : Exécution d'un bloc de code par un (et un seul) thread + synchronisation,
- MASTER : Exécution d'un bloc de code par le thread 0 (maître) + synchronisation,
- WORKSHARE (fortran) : Répartition de fonctions intrinsèques portant sur des tableaux (exemples: MATMUL, DOT_PRODUCT, SUM, COUNT, etc ...).

Notions de tâches avec OpenMP

Openmp autorise une gestion par tâche avec la possibilité de créer un pool de tâches qui seront collectées et exécutées par les différents threads de la région.



Notions de tâches avec OpenMP

Tâche OpenMP : un "travail" à réaliser, indépendant, du code exécutable et des données associées, affectés à un thread.
Nous avons déja rencontré ces tâches OpenMP, crées implicitement dans une région parallèle

```
#pragma omp parallel num_threads(3)
{
  // Création de trois tâches implicites, affectées à trois threads
}
```

ou encore dans un un bloc sections où chaque section correspond à une tâche.
Pour créer une tâche explicite, il faut utiliser la directive task

```
#pragma omp parallel
{
  ...
  #pragma omp task
  {
    // Code à effectuer par la tâche.
  }
```

Construction de tâches : directive TASK

- Dès qu'un thread rencontre cette directive, il crée la tâche.
- Le pool de tâches créées existe pour chaque région parallèle : le thread qui crée la tâche ne sera pas nécessairement celui qui l'exécute (sauf en cas de clause tied).
- Son exécution peut être immédiate ou différée, selon la disponibilité des threads (et les clauses).
- En fin de région parallèle, une synchronisation implicite assure que toutes les tâches seront complètes.

Quelques clauses de la la directive TASK :

- if (expression) : si l'expression est vraie, la tâche est exécutée immédiatement par le thread sinon mise dans le pool
- untied : tout thread peut reprendre la tâche si elle est suspendue.
- priority(N) : défini un ordre de grandeur de la priorité de la tâche.

Exemple de construction de tâches

```
#pragma omp parallel default(none) shared(nb_tasks)
{
// Le thread master est chargé de la création des tâches
#pragma omp master
for(int i=0;i<nb_tasks;++i)
 {
   #pragma omp task
   do_task(i); // do_task une fonction définie par ailleurs
 }

 autres_calculs();

}
```

Dès qu'un thread de la région sera disponible, il exécutera une des tâches do_task(i)

Construction de tâches : exemple fortran

```
real(kind=8),dimension(n)::a,b,c
a(:)=10. ; b(:)=20. ; c(:)=0.

!$omp parallel default(none) shared(a,b,c)

! Création des tâches par un seul thread
!$omp single
 print *,'Print rang après single ',omp_get_thread_num()
 do i=1,100
  !$omp task untied
  print *,'Tache num ',i,'sur le thread',omp_get_thread_num
  c[i]=a[i] + b[i]
  !$omp end task
  end do
  !$omp end single nowait
  print *,'rang',omp_get_thread_num(), 'Somme finale',c(1)
!$omp end parallel
```

```
Print rang après single      0

Tache num 1 sur le thread  0
Tache num 2 sur le thread  1
Tache num 3 sur le thread  3
Tache num 4 sur le thread  2
Tache num 5 sur le thread  2
Tache num 6 sur le thread  3
Tache num 7 sur le thread  1
Tache num 8 sur le thread  0
Tache num 9 sur le thread  0
Tache num 10 sur le thread 1
....
rang       2 Somme finale    30.0
rang       3 Somme finale    30.0
rang       0 Somme finale    30.0
rang       1 Somme finale    30.0
```

Directive TASKWAIT

- Spécifie un point d'attente de terminaison des tâches créées avant l'appel à la directive
- Il s'agit d'une barrière spécifique aux tâches.
- Dans l'exemple ci-dessous : la tache 4 sera effectuée après les taches 1, 2 et 3

```
!$omp parallel
!$omp single

!$omp task
   printf("Task 1\n");
!$omp task
   printf("Task 2\n");
!$omp task
   printf("Task 3\n");

!$omp taskwait

!$omp task
   printf("Task 4!\n");
```

Résumons ...

Gestion des tâches

- Directive TAKS : permet de créer un pool de tâches indépendantes,
- Permet d'utiliser des threads en "tâche de fond" qui exécute une ou plusieurs tâches,
- Exécution des tâches immédiate ou différée,
- Synchronisation en fin de zone parallèle.

Synchronisation

En mémoire partagée, il peut être nécessaire d'introduire une synchronisation entre les tâches. Cela évite par exemple la modification d'une valeur partagée dans un ordre aléatoire (race condition)
Nous avons vu que de nombreuses directives impliquent une synchronisation implicite en fin de région (parallel , sections ...).
Il existe d'autres moyens de gérer la synchronisation :

- Directives de synchronisation explicites : exemple BARRIER
- Directives qui permettent la synchronisation de données : FLUSH
- Directives qui permettent d'exécuter des portions de code dans un certain ordre pour aboutir à une forme de synchronisation : ORDERED
- Directives ATOMIC et CRITICAL
- Les routines spécifiques de la bibliothèque

directive BARRIER

Barrière de synchronisation : chaque threads attend que tous les autres aient terminé leur travail.

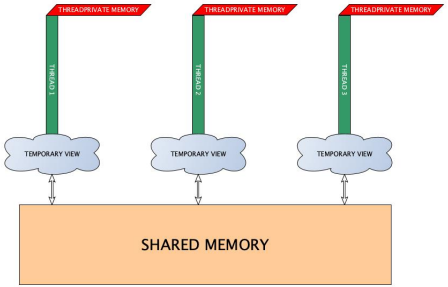
- La directive doit être exécutée par tous les threads avant de passer à la tâche suivante.
- ⚠ : si le code est écrit de telle façon que tous les threads ne soient pas en mesure d'exécuter l'instruction, possibilité de *deadlock*, le programme se bloque en cours d'exécution.

Exemple :

```
!$OMP PARALLEL
  if (mythreadnum < 5) then
     !$OMP BARRIER
  endif
!$OMP END PARALLEL
```

Flush

Rappel : gestion de la mémoire par les threads



Du fait de la temporary view, la mémoire vue par le thread n'est pas nécessairement consistante avec la mémoire partagée.

Flush : opération qui rend la mémoire temporaire d'un thread consistante avec la mémoire partagée.

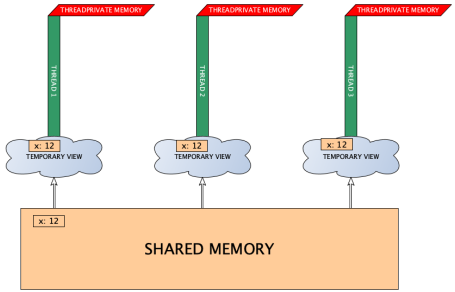
Flush

Flush : opération qui rend la mémoire temporaire d'un thread consistante avec la mémoire partagée.

- Assure que le contenu en tmp est bien celui en shared (e.g. en cas d'écriture en shared par un autre thread).
- Assure que le contenu en shared est bien celui de la dernière écriture en tmp sur le thread courant.
- ⚠ : ne concerne que le thread qui flush ! Ce n'est pas une synchronisation complète !

Exemple

```
#pragma omp flush(x)
```



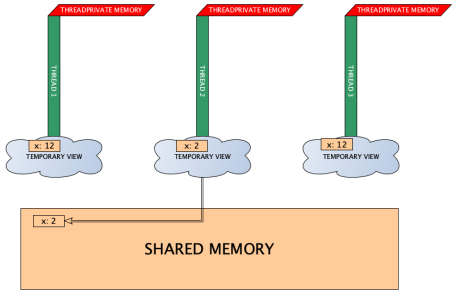
Flush

Flush : opération qui rend la mémoire temporaire d'un thread consistante avec la mémoire partagée.

- Assure que le contenu en tmp est bien celui en shared (e.g. en cas d'écriture en shared par un autre thread).
- Assure que le contenu en shared est bien celui de la dernière écriture en tmp sur le thread courant.
- ⚠ : ne concerne que le thread qui flush ! Ce n'est pas une synchronisation complète !

Exemple

```
#pragma omp flush(x)
```



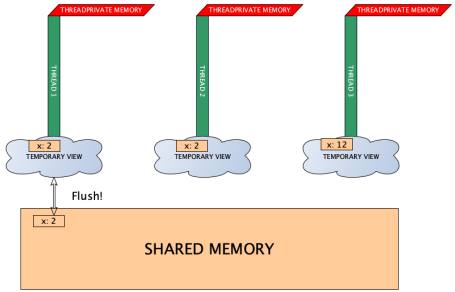
Flush

Flush : opération qui rend la mémoire temporaire d'un thread consistante avec la mémoire partagée.

- Assure que le contenu en tmp est bien celui en shared (e.g. en cas d'écriture en shared par un autre thread).
- Assure que le contenu en shared est bien celui de la dernière écriture en tmp sur le thread courant.
- ⚠ : ne concerne que le thread qui flush ! Ce n'est pas une synchronisation complète !

Exemple

```
#pragma omp flush(x)
```



Flush

Quand flusher ?

- Avant une lecture de variable modifiée par un autre thread
- Après une écriture d'une variable susceptible d'être lue par un autre thread

Mais : utiliser flush pour la gestion de la synchronisation des variables est compliquée et sources de bugs. Mieux vaut utiliser les "vraies" directives de synchronisation.

Directive et clause ORDERED

- Dans le cas d'un debogage par exemple on peut vouloir exécuter une boucle de façon ordonnée
- On utilise la clause ORDERED avec la directive DO : l'ordre des itérations sera identique à celui d'une exécution séquentielle.
- Les instructions effectuées de façon séquentielle sont encadrées par la directive ORDERED.

```
integer :: i,rang
! $OMP PARALLEL DEFAULT(PRIVATE)
rang = OMP_GET_THREAD_NUM ()
! $OMP DO SCHEDULE(RUNTIME) ORDERED
do i = 1, 9
!$OMP ORDERED
print *,"Rang :",rang,"; iteration :",i
!$OMP END ORDERED
end do
! $OMP END DO NOWAIT
!$OMP END PARALLEL
```

Exécution avec 4 threads :

```
> export OMP_SCHEDULE="STATIC,2"
> export OMP_NUM_THREADS=4 ; a.out
Rang : 0 ; iteration : 1
Rang : 0 ; iteration : 2
Rang : 1 ; iteration : 3
Rang : 1 ; iteration : 4
Rang : 2 ; iteration : 5
Rang : 2 ; iteration : 6
Rang : 3 ; iteration : 7
Rang : 3 ; iteration : 8
Rang : 0 ; iteration : 9
```

Directive ATOMIC

ATOMIC atomic-clause assure qu'une variable partagée dans une région parallèle, n'est lue et modifiée que par un seul thread à la fois. atomic-clause = read/write/capture/update

```
integer :: compteur, rang
compteur = 92290
! $OMP PARALLEL PRIVATE(rang)
rang = OMP_GET_THREAD_NUM ()
!$OMP ATOMIC update
compteur = compteur + 1 ! Un seul accès à la fois !
print *,"Rang :",rang, "; compteur vaut :",compteur
!$OMP END PARALLEL
print *,"Au total, compteur vaut :", compteur
end program parallel
```

```
Rang : 1 ; compteur vaut : 92291
Rang : 0 ; compteur vaut : 92292
Rang : 2 ; compteur vaut : 92293
Rang : 3 ; compteur vaut : 92294
Au total, compteur vaut : 92294
```

- Concerne l'instruction qui suit immédiatement la directive (pas un bloc de code).
- Les types d'instructions autorisés dépendent de la valeur d'atomic-clause. Voir la norme.
- Il n'y a pas de notion d'ordre pour l'exécution.

Directive CRITICAL

Définition d'une region "critique" qui ne sera exécutée que par un seul thread à la fois.
⚠ : différent de single !

```
#pragma omp parallel for private(b), shared(res)
for(i=1;i<niters;i++)
{
  b = doit(i)
  #pragma omp critical
  { // Région critique
    combine (b, res)
  }
}
```

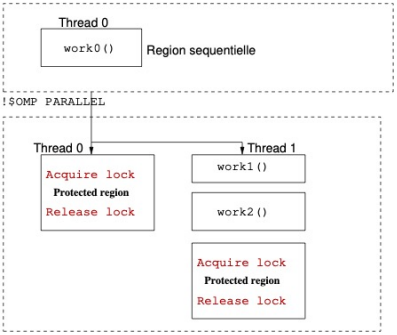
- Ordre d'exécution indéterminé.
- Possibilité de nommer (label) les régions critiques.
- Etend la notion de la directive ATOMIC à un bloc d'instructions.
- Cependant, pour une opération sur une variable, ATOMIC offre de biens meilleures performances.

Bibliothèque : les "Lock Routines"

Une autre façon de synchroniser : faire appel à des routines de la bibliothèque OpenMP. Ces routines permettent de modifier la valeur des variables "locks" (elle sont d'un type particulier). On peut les considérer comme un drapeau à mettre ou à enlever. Chaque thread *consulte* l'état du drapeau et agit en conséquences. Il peut aussi l'ignorer.

- OMP_init_lock : initialisation d'un drapeau de lock
- OMP_set_lock : active le drapeau de lock
- OMP_unset_lock : désactive le drapeau de lock
- OMP_test_lock : consulte l'état du drapeau de lock
- OMP_destroy_lock : supprime le drapeau de lock
- Alternatives plus flexibles à atomic et critical
- Préférer atomic ou critical quand c'est possible

Bibliothèque : les “Lock Routines”




```
omp_lock_t lck;
omp_init_lock(&lock1);
#pragma omp parallel shared(lock1) private(tmp,tmp2,tmp3,id)
{
  id = omp_get_thread_num();
  if (rang==0)
  {
    omp_set_lock(&lock1);
    tmp = do_some_work(id);
    omp_unset_lock(&lock1);
    printf("%d %d \n", id, tmp);
  }
  else if(rang==1)
  {
    tmp2 = do_other_work(id);
    omp_test_lock (&lck1);
    tmp3 = do_third_work(id);
    omp_unset_lock(&lock1);
   }
}
omp_destroy_lock(&lock1);
```

Cela permet donc des synchronisations très fines en fonction des tâches à exécuter et offre beaucoup de flexibilité mais à manipuler avec précautions !

Résumé des synchronisations explicites/implicites

- Synchronisations explicites
  - !$*OMP* BARRIER
  - !$*OMP* PARALLEL et !$*OMP* PARALLEL DO, !$*OMP* PARALLEL SECTIONS, PARALLEL WORKSHARE
- Synchronisations implicites (clause NOWAIT)
  - !$*OMP* DO
  - !$*OMP* SECTIONS
  - !$*OMP* SINGLE
  - !$*OMP* WORKSHARE
- Autres directives de synchronisations
  - Synchronisation explicite des données : directive FLUSH,
  - Ordonnancement des tâches : directives ORDERED, ATOMIC, CRITICAL.

Exercice

Rendez-vous dans le projet habituel pour la fin des exercices.

Optimiser son code parallélisé avec OpenMP : les grands principes

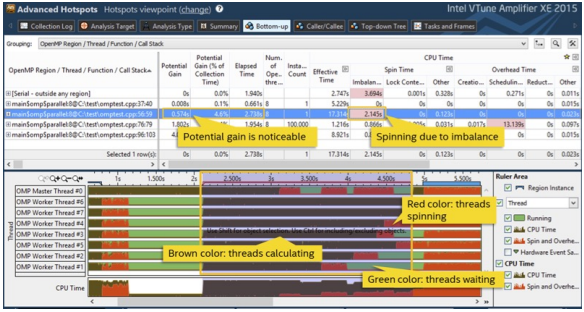
Le code séquentiel doit déjà être optimisé !

- Optimisation des zones parallèles par rapport au nombre de threads disponibles et à la quantité de travail par thread (overheads)
  - Minimiser le nombre de zones parallèles
  - Limiter l'usage des barrières de synchronisation
  - La directive PARALLEL FOR contient une barrière de synchronisation : mieux vaut plusieurs for
  - Utiliser la clause SCHEDULE(RUNTIME) pour pouvoir adapter son code à l'environnement de calcul
- Eviter le false-sharing : intervention de différents threads sur des données contigües en mémoire (essentiellement une boucle). Ces données sont chargées dans la ligne de cache par plusieurs threads à la fois, on a donc une perte de cohérence du point de vue de la mémoire partagée, ce qui induit une forte chute des performances attendues.
  La taille du paquet d'itérations traitées par thread doit être suffisante.

Outils de profiling

Utiliser un outil de profiling permet :

- une analyse des tâches thread par thread
- d'identifier les threads qui attendent/calculent, etc ...



Exemples d'outils permettant de de faire du profiling avec OpenMP :

- Vtune Amplifier XE (payant mais il existe une version académique)
- Scalasca, Tau, ...

Les avantages d'OpenMP

- Facile à mettre en oeuvre, peu intrusif,
- Programmation incrémentale, progressive
- Gestion des threads transparente et portable
- Supporté par beaucoup d'OS et de compilateurs
- Pas de gestion explicite des communications.

Les limites d'OpenMP

- Limité à un type d'architecture (mémoire partagée).
- Exécution limitée par la taille du nœud à mémoire partagée sur lequel il s'exécute.
- La bande passante mémoire cumulée à l'intérieur d'un nœud SMP peut encore plus limiter le nombre de cœurs utilisables efficacement
- Debugging ardu (localité des données ...)
- Très facile d'obtenir de mauvaises performances ...

Notions non abordées dans ce cours

- Affinités des threads
- Affinité mémoire
- Notions de synchronisation point à point
- Apports d'OpenMP 4 (SIMD, targets ...)

1 OpenMP

2 GPGPU

Introduction au GPGPU

[Introduction au calcul parallèle]

Frédéric Audra Glenn Cougoulat

Mars 2023

Collège des écoles doctorales

Plan du cours

- Le contexte
  - Historique
  - L'évolution de la pile logicielle
  - L'évolution du matériel
- L'architecture
  - Rappel: architecture CPU
  - Architecture GPU
  - CPU vs GPU
- Environnements GPU à GRICAD
  - Présentation des ressources
  - Utilisation du cluster GPU
- Introduction à CUDA
  - Séquençage d'un code CUDA
  - Compilation avec `nvcc`
- Introduction à CUDA (suite)
  - Les threads: processus et parallèlisme
  - Visualiser l'utilisation du GPU
- Introduction à OpenACC
  - Origine, définition
  - Syntaxe
  - Directives
  - Clauses et niveaux de parallélisme
  - Synthèse

Le contexte

Historique

A l'origine de leur création, les cartes graphiques (GPU) avaient pour but d'accélérer le pipeline de rendu[1] des scènes 3D. Les premières générations de GPU implémentaient tout ou partie de ce pipeline de manière hardware. L'histoire des GPU connait un tournant majeur avec l'introduction en 2001 d'un pipeline programmable. Cette évolution de la pile logicielle aboutie en 2007 à la naissance du premier kit de développement CUDA et au début de l'ère du :

GPGPU:General **P**urpose **C**omputing on **G**raphics **P**rocessing **U**nits.

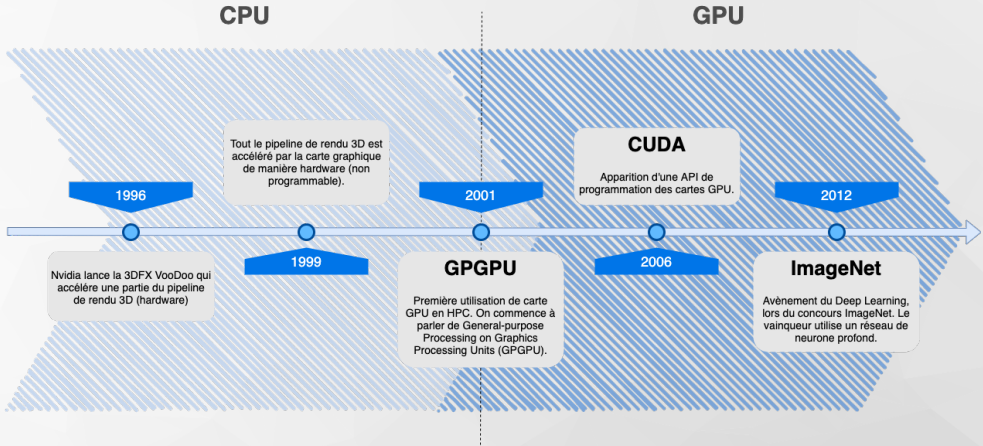
On commence alors à exploiter la puissance de calcul des GPUs en calcul scientifique, pour le traitement des tâches massivement parallèles.

[1] Ensemble des étapes de création d'une scène 3D, du chargement des données, au dessin en passant par la rasterisation et l'application de textures.

L'évolution de la pile logicielle

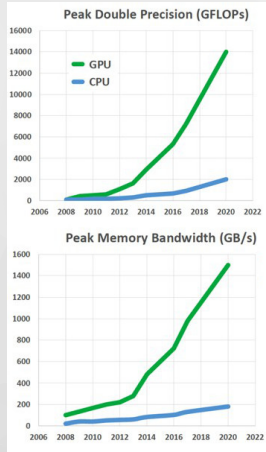
- **Avant 1995 :** les rendus 3D sont calculés sur CPU. Lenteur, et possiblités trés limitées.
- **1996 :** Diamond sort la 3Dfx VooDoo (premier succès 50 à 80% du marché, accélération matérielle d'une partie du pipeline)
- **1999 :** Nvidia lance les premières cartes "GPU" (GeForce 256) qui accélèrent l'intégralité du pipeline de rendu de manière hardware (non-programmable). Augmentation des résolutions d'écrans, et de la complexité des modèles 3D (nombre de triangles).
- **2001 :** Avec la GeForce 3, Nvidia introduit le pipeline programmable via une fonction "shader". Début de l'ère des GPGPU. Premières utilisations pour le calcul scientifique, programmation trés complexe.
- **2006 :** Nvidia introduit la GeForce 8800 et sort une API dédiée au calcul scientifique (programmation en C). Naissance de CUDA.

L'évolution de la pile logicielle: chronologie, naissance du GPGPU



L'évolution du matériel: une affaire de design

Contrairement à l'évolution des CPUs (Central Processing Unit) qui à longtemps obéit la loi de Moore[2], les GPUs suivent une évolution bien plus importante, qui s'explique par la conception même de ce type de cartes. Nous verrons ces différences de conception dans la partie suivantes sur les architectures.



[2]Constatation selon laquelle le nombre de transistors d'un CPU double tous les 2 ans.

L'évolution du matériel: les différentes gammes de GPU (Nvidia)

S'il existe d'autres constructeurs de cartes GPU (AMD, Intel, etc...), nous nous focaliserons sur les cartes Nvidia (utilisées sur les machines de GRICAD) et en particulier les V100 et A100 de la gamme Tesla conçues spécifiquement pour l'IA et le HPC.

- GeForce : cartes graphiques pour usage grand public : jeux, infographie, calcul parallèle
- Quadro : cartes graphiques dédiées aux professionnels : création de contenu digital, CAO, DAO
- **Tesla : cartes graphiques dédiées uniquement au calcul scientifique**
- Tegra : liées aux systèmes mobiles (smartphones, tablettes)

L'évolution du matériel: Nvidia Tesla A100

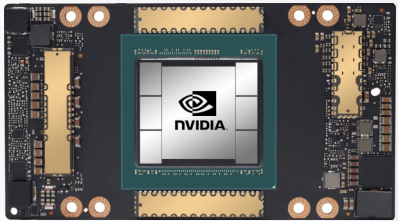


Figure 1: Nvidia Tesla A100 (PCIe version)

- Pas de sortie vidéo
- Spécifiquement conçues pour le calcul scientifique
- Double précision
- Plus de mémoire embarquée (jusqu'à 80Go)
- Possibilité de communication « Infiniband »[3] pour connecter des GPU embarquées sur différents ordinateurs.
- Possibilté de communication NvLink (SXM2) pour connecter plusieurs GPU embarquées sur un même noeud.
- Possibilité de spliter le GPU en Multi-Instances GPU (MIG : partitions virtuelles de GPU indépendantes)

[3] Réseau à haut débit et faible latence.

Le contexte : pourquoi le GPU ?

Le monde du calcul et de la simulation numérique en particulier est en permanence confronté à une complexification des modèles, une augmentation de la taille des problèmes, à l'apparition de nouvelles disciplines. Les besoins en simulation numérique augmentent sans cesse dans des domaines toujours plus variés (médecine, climat, sciences de la communication, nucléaire, nanobiologie, etc...).

Pour comprendre pourquoi le GPU à pris une large importance dans le paysage de la simulation numérique, il est important de comparer ces deux architectures pour bien cerner les limitations et les capacités de chacunes d'entres elles.

L’architecture

Rappel: architecture CPU

Ci-contre le schéma de l'architecture CPU que vous avez étudié précédement.
Nous reverrons donc rapidement les notions essentielles de cette architecture.

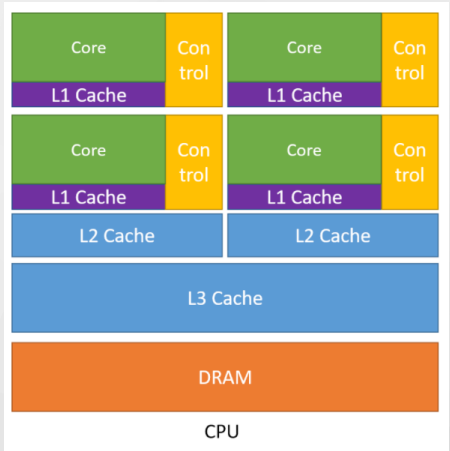


Figure 2: Architecture CPU multi-coeurs

Architecture CPU : un design orienté *latence*

Le but de cette architecture est d'optimiser la performance séquentielle pure en diminuant la latence des accès mémoire. Même s'il existe, comme vous l'avez vu précédement, différentes techniques de parallélisation, ce type d'architecture est particulièrement indiqué dans des traitements séquentiels.

- fréquence d'horloge élevée pour effectuer plus d'opérations pour éviter d'attendre
- différents niveaux de mémoires cache (à faible latence) pour réduire et maitriser la latence des accès à la RAM du système
- système de contrôle sophistiqué (prédiction de branchement, exécution out-of-order, etc...) pour l'optimisation de l'accès aux données

Architecture CPU : limitations

L'architecture CPU confrontée au besoin constant d'augmentation des performances de calcul touche aujourd'hui à des limitations physiques :

- **Consommation électrique:** elle est proportionnelle à la fréquence d'horloge du processeur et augmente de façon exponentielle. Cela est donc un frein considérable à l'heure où on multiplie les coeurs de calcul dans les clusters et où l'efficacité énergétique est un enjeu majeur.
- **Dissipation thermique:** elle est là encore liée à l'augmentation de la fréquence. Celà pose donc des problèmes de gestion du refroidissement au niveau des CPU et de l'ensemble des infrastructures.
- **Finesse de gravure des composants:** de 32nm en 2010 à 5nm en 2020, les constructeurs repoussent toujours plus loin les limites physiques. On peut se demander jusqu'à quand cela sera possible ?

Architecture GPU : un design orienté *débit*

La différence de stratégie entre les 2 architectures est visible sur ce schéma. Alors qu'un CPU est constitué de quelques dizaines de coeurs, le GPU embarquent plusieurs milliers de coeurs. Les coeurs d'un GPU disposent de beaucoup moins de cache que sur un CPU, et le controle se limite à la gestion des threads. La fréquence d'horloge est moins élevée sur un architecture GPU (~1500MHz). L'objectif de cette architecture SIMD est de favoriser le traitement de plusieurs threads à la fois ce qui permet de masquer les latences dues à la mémoire. Ce design est orienté *debit*.

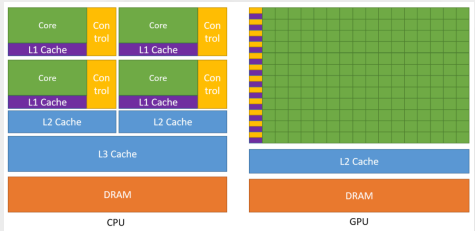


Figure 3: Architecture CPU vs GPU

Les différentes architectures : bilan

CPU :

Applications

- applications généralistes

Concept

- optimisé pour qu'un thread finisse le plus rapidement possible
- pipelining important
- gros caches
- logique complexe (prédiction de branchement, prefetching, out-of-order, hyper-threading)
- parallélisation hiérarchique : thread/SIMD

GPU

Applications

- calcul flottant

Concept

- optimisé pour qu'un groupe de threads finissent rapidement
- context-switching
- petit caches, beaucoup de registres
- logique réduite à la gestion des threads
- parallélisation à 1 seul niveau : threads

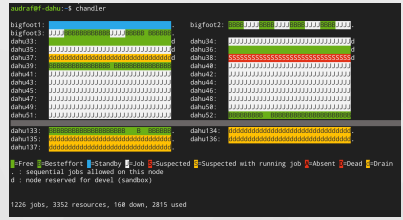
Environnements GPU à GRICAD

Présentation des ressources : les premiers noeuds GPU ‘Bigfoot’

Les premiers noeuds ‘bigfoot’ équipés d’accélérateurs GPU, précédement hébergés dans le cluster Dahu, ont rejoint le cluster GPGPU Bigfoot en 2022. Bigfoot est donc aujourd’hui composé de noeuds de types différents :

Noeuds à base de V100 :

- CPUs: 2 x Xeon Gold 6130 2.1Ghz (16c) soit 32 coeurs ou 2 x Xeon Gold 5218R 2.10GHz (20c) soit 40 coeurs
- RAM: 192Go
- GPUs: 4 x Nvidia Tesla V100 32Go HBM2 par noeud
- Interconnection des GPU via le bus NvLink : 300GB/s
- Interconnection des noeuds via le réseau OmniPath : 100GB/s



☞ Noeuds à privilégier pour les projets qui veulent exploiter le multi-GPU grâce à la forte bande passante du SXM2.

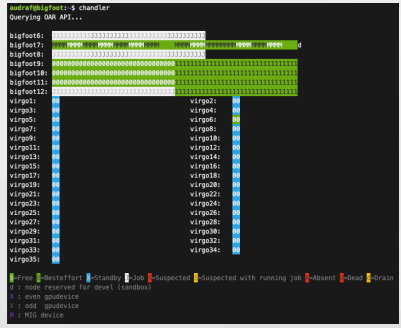
Présentation des ressources : le cluster IA/DeepLearning 'Bigfoot'

Noeuds à base de A100 :

- CPUs: 2 x AMD EPYC 7452 1,5GHz soit 64 coeurs
- RAM: 192Go
- Connection inter-noeuds via OmniPath : 100GB/s
- 2 types de configuration GPU pour ces noeuds :
  - 2 x Nvidia Tesla A100 40Go, interconnexion PCIe 64GB/s
  - 2 x Nvidia Tesla A100 40Go, interconnexion PCIe, partitionnées en 2 x 7 MIG (Multi-Instances GPU)

Noeuds 'Virgo' (GPU légers) : - 1 x Nvidia Tesla T4 32Go, disponibles via les noeuds virgo* entre 00h et 06h grâce au projet FALL (VM UGA).

👉 Syntaxe OAR : `-p "fall='NO'"` ou `-p "fall='YES'"`



Présentation des ressources : les GPUs Tesla V100

Avec 5120 coeurs et une puissance théorique de 7,8TFlops (en double précision) les V100 qui équipent les noeuds Bigfoot offrent encore aujourd'hui[4] un niveau de performance élevé. Il est intéressant de noter que le super calculateur Jean Zay opère les mêmes GPUs.

Les GPUs V100 de Bigfoot sont interconnectées via un bus NvLink (sxm2). Les spécifications ci-contre, montrent les différences entre les modèles au sein de cette gamme de GPU.

La bande passante du bus Nvlink (x10) permet un gain significatif de performance (par rapport au PCIe). Les GPUs ainsi interconnectés sont donc adaptés aux jobs multi-GPU.

SPECIFICATIONS

	V100 PCIe	V100 SXM2	V100S PCIe
GPU Architecture		NVIDIA Volta	
NVIDIA Tensor Cores		640	
NVIDIA CUDA® Cores		5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	8.2 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	16.4 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS	130 TFLOPS
GPU Memory	32 GB /16 GB HBM2		32 GB HBM2
Memory Bandwidth	900 GB/sec		1134 GB/sec
ECC		Yes	
Interconnect Bandwidth	32 GB/sec	300 GB/sec	32 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink™	PCIe Gen3
Form Factor	PCIe Full Height/Length	SXM2	PCIe Full Height/Length
Max Power Comsumption	250 W	300 W	250 W
Thermal Solution		Passive	
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC®		

[4] En 2020. La sortie des modèles A100 vient cependant repousser les limites actuelles.

Présentation des ressources : les GPUs Tesla A100

Les Tesla A100 qui équipent les noeuds du cluster Bigfoot comptent **8192 coeurs** CUDA pour une puissance théorique de 9,7TFlops (en double précision). Il est intéressant de noter que le super calculateur Jean Zay opère les mêmes GPUs.

Les GPUs A100 de Bigfoot sont interconnectées via le bus PCI-e (64Go/s). Les spécifications ci-contre, montrent les différences entre les modèles au sein de cette gamme de GPU.

Pour aller plus loin sur l'architecture Ampere :

- Ampère architecture whitepaper.

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64	9.7 TFLOPS			
FP64 Tensor Core	19.5 TFLOPS			
FP32	19.5 TFLOPS			
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*			
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*			
INT8 Tensor Core	624 TOPS | 1248 TOPS*			
GPU Memory	40GB HBM2	80GB HBM2e	40GB HBM2	80GB HBM2e
GPU Memory Bandwidth	1,555GB/s	1,935GB/s	1,555GB/s	2,039GB/s
Max Thermal Design Power (TDP)	250W	300W	400W	400W
Multi-Instance GPU	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 5GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe		SXM	
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600GB/s ** PCIe Gen4: 64GB/s		NVLink: 600GB/s PCIe Gen4: 64GB/s	
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs		NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs	

* With sparsity

** SXM4 GPUs via HGX A100 server boards; PCIe GPUs via NVLink Bridge for up to two GPUs

Utilisation des ressources GPU

Avant d'exploiter la puissance des GPUs, il est important d'avoir à l'esprit les principales étapes pour y parvenir. Il faudra donc :

1. activer la couche logicielle nécessaire à l'interaction avec les GPUs (Kit de développement), sauf dans le cas d'utilisation d'un container qui embarque cette couche (images issues du cloud Nvidia NGC notamment).
2. activer le gestionnaire d'environnements adapté à vos besoins/critères (reproductibilté, facilité de mise en oeuvre, etc...)
3. activer l'environnement adéquat pour utiliser les bibliothèques choisies (lancer le container qui intègre les bibliothèques dont vous avez besoin)

Utilisation des ressources GPU : rappel gestion de l'environnement

Plusieurs gestionnaires d'environnements sont disponibles sur les machines de GRICAD. Dans ce module, nous parlerons uniquement des gestionnaires d'environnements préconisés pour l'usage des GPUs (Conda et Singularity). Ci-dessous un bref rappel des commandes pour activer `Conda`.

Pour activer `Conda` :

```
audraf@dahu:~/ source /applis/environments/conda.sh
```

Pour utiliser les containers via `singularity`, utiliser la version présente sur les noeuds :

```
audraf@dahu:~/ singularity shell --nv chemin_vers_mon_image.simg
```

Utilisation des ressources GPU : choisir son gestionnaire d'environnements

Le prêt à l'emploi avec Conda :

La façon la plus simple et directe pour utiliser les GPUs consiste à activer les environnements fournis par GRICAD via `Conda`. Pour cela, il est intéressant de parcourir la documentation. Plusieurs environnements sont à votre disposition, ils couvrent un champs d'application assez large bien qu'orienté IA. Vous y trouverez notamment les principaux frameworks python dans ce domaine (Tensorflow, PyTorch, etc...).

La portabilité et la reproductibilité avec les containers :

L'utilisation de containers grâce à `singularity` permet, entre autres, de bénéficier d'une bibliothèque complète d'images maintenues par Nvidia (NGC). Selon les besoins de vos projets, l'utilisation de telles images peut vous permettre un réel gain de temps. L'isolation complète des bibliothèques et du code dans un container permet un bon degré de reproductibilité.

Les containers lancés via singularity ont généralement accès à vos données sur `/home` et sur `/bettik`, et il est aussi possible de compléter les montages selon vos besoins.

Utilisation des ressources GPU : réservation OAR

L'exploitation des noeuds GPU nécessite une syntaxe particulière au niveau de la réservation de ressources avec **OAR**. Il est en effet indispensable de préciser au gestionnaire de Jobs de quelles ressources GPU vous avez besoin pour votre job.

Les ressources GPU sont identifiées par les mots clé : `gpu` et `migdevice`.

Vous pouvez utiliser ces mots clé pour réserver vos ressources :

```
# Réserver 1 GPU
> oarsub -I -l /node=1/gpu=1/core=10

# Réserver 1 modèle de GPU en particulier
> oarsub -I -l /node=1/gpu=1/core=10 -p "gpumodel='A100'"
```

Pour plus de détails, consultez la documentation à propos de la réservation de ressources GPU.

Utilisation des ressources GPU : réservation OAR

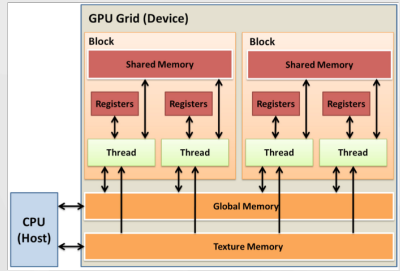
```
# Accès aux MIG
> oarsub -I -l /node=1/gpu=1/migdevice=1/core=4 -t devel

# Réserver 1 GPU T4
> oarsub -I -l /node=1/gpu=1 -p "fall='YES'"
```

Attention 👉 :pour les raisons explicitées plus haut dans ce cours (architecture), il est nécessaire de prêter attention aux ressources CPU demandées lors de jobs GPU. Elles doivent en effet être en adéquation avec les ressources GPU pour ne pas créer de goulot d'étranglement (/migdevice=1/core=4).

Introduction à CUDA

Séquençage d'un code CUDA

1. Le CPU alloue les ressources nécessaires pour l'execution sur le GPU
2. Le CPU copie les données dans la mémoire globale du GPU
3. Le GPU execute le « kernel »
4. Le CPU copie les résultats depuis la mémoire globale du GPU



Séquençage d'un code CUDA

Définition des fonctions et variables en CUDA:

```
__global__ void KernelFunc(…);  // Kernel accessible du CPU (Host)
__device__ void DeviceFunc(…);  // Fonction accessible sur le GPU
__device__ int GlobalVar;       // Variable dans le GPU
__shared__ int SharedVar;       // Variable partagée par Thread dans le bloc
__host__ void HostFunc(…);      // fonction appelable sur le CPU
```

Variables des registres (lecture seule) : `threadIdx`, `blockDim`, `blockIdx`

Exemple de code CUDA

```
__global__ void mykernel(void) {     // Déclaration d'un kernel sur le CPU
}
int main (void) {
    mykernel<<<1,1>>>();             // Attribution d'un thread dans un bloc
    printf("Hello World!\n");
    return 0;
}
```

Compilation et exécution :

```
$ nvcc main.cu
$ ./a.out
Hello World!
```

Compilation avec `nvcc`

Documentation utile

☞ La documentation en ligne est disponible ici.

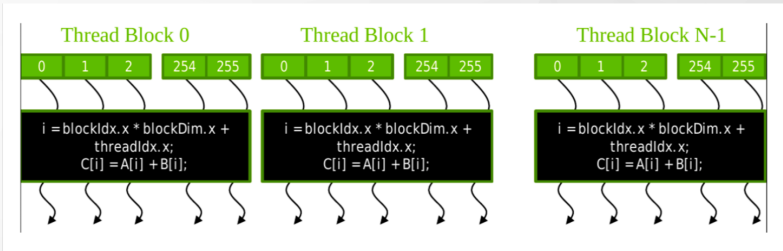
Commande complexe nécessitant de connaitre l'execution de son programme de l'environnement de librairies disponibles et de l'architecture GPU

Exemples d'options importantes :

- Utilisation des librairies de tests incluant le calcul de timing `–library mylibrary`
- Nombre maximum de threads à paralléliser `–Threads number`
- Architecture GPU `--gpu-architecture` par défaut `sm_50`

Les threads: processus et parallèlisme

- Chaque kernel Cuda est exécuté par une grille de threads
- Chaque thread est indexé pour calculer les adresses mémoires
- Les theads de différents blocs ne peuvent interagir
- Tous les threads d'une grille exécute le même noyau



Visualiser l'utilisation du GPU

```
audraf@bigfoot1:~$ nvidia-smi
Mon Feb  1 11:05:02 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.80.02    Driver Version: 450.80.02    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:1A:00.0 Off |                    0 |
| N/A   41C    P0    57W / 300W |      0MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:1C:00.0 Off |                    0 |
| N/A   35C    P0    56W / 300W |      0MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:1D:00.0 Off |                    0 |
| N/A   36C    P0    56W / 300W |      0MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:1E:00.0 Off |                    0 |
| N/A   41C    P0    59W / 300W |      0MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
audraf@bigfoot1:~$
```

La commande `nvidia-smi` permet d'interagir avec le driver pour maîtriser et visualiser l'utilisation du GPU.

Le slide suivant liste quelques unes des options intéressantes de `nvidia-smi`.

Visualiser l'utilisation du GPU

Quelques commandes nvidia-smi intéressantes :

- Visualiser la charge du GPU #3:

```
nvidia-smi --query-gpu=utilization.gpu,utilization.memory \
           --format=csv --loop-ms=1000 -i 3
```

- Visualiser l'ensemble des propriétés d'un GPU (#3 par exemple):

```
nvidia-smi -q -i 3
```

- Monitorer l'exécution d'un processus sur un GPU:

```
nvidia-smi --query-compute-apps=pid,process_name,used_memory --format=csv \
           -i 3
```

- Pour plus de détails sur toutes les options de cette commande : `nvidia-smi -h`

Introduction à OpenACC

Origine, définition

Origine :

Né en 2011 et développé par le consortium : **Cray**, **Nvidia**, **CAPS**, et **PGI** ; OpenACC est un standard de programmation qui permet le calcul parallèle sur des **systèmes hétérogènes CPU/GPGPU.**

Définition :

- A l'instar d'OpenMP, OpenACC fonctionne grâce à l'ajout de commandes dans du code **C/C++** ou **Fortran**.
- OpenACC utilise ces directives pour indiquer au compilateur où et comment paralléliser les boucles, et où et comment gérer les données entre des mémoires d'hôte et d'accélérateur potentiellement séparées.
- OpenACC est pris en charge par les compilateurs de PGI *(>=12.6)*, Cray et CAPS *(>=3.1)*, GCC (support partiel).

Syntaxe des directives et clauses

Syntaxe C :

```
#pragma acc directive-name[clause[[,]clause]...]
{
    ...
}
```

Syntaxe Fortran :

```
!$acc directive-name[clause[[,]clause]...]
    ...
!$acc end directive-name
```

Directives

Directives importantes pour le parallèlisme :

- **parallel loop** : (*prescriptive*) l'intégrité du code parallélisé revient au développeur, qui a plus de contrôle sur l'optimisation. -> *Fonctionnement identique à OpenMP.*
- **kernels** : (*descriptive*) le compilateur se charge d'analyser le code à paralléliser et garantir son intégrité. Il est également libre d'optimiser le code. Une seule directive kernels peut s'appliquer à une grande portion de code.

Les deux approches sont valides et leur performance est comparable.

Directives pour la gestion des données :

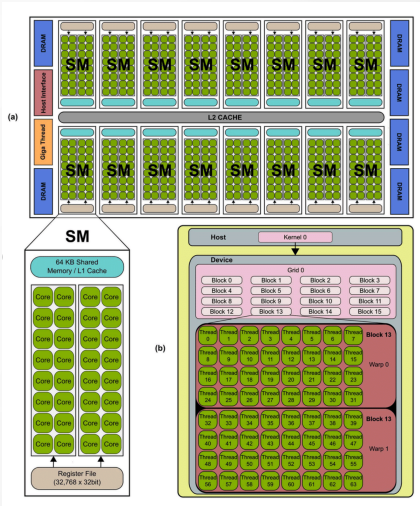
- **data** : (clauses: create, copy, pcopy, copyin...)

Clauses et niveaux de parallélisme :

OpenACC permet de contrôler l'optimisation du parallélisme en spécifiant le niveau de parallélisme grâce aux clauses : **Gangs**, **Workers** et **Vectors**.

```
#pragma acc parallel, num_gangs[32], num_workers[256]
{
#pragma acc loop gang
for(i=0; i < n; i++) {...}
...
}
```

Clauses et niveaux de parallélisme :



A ces niveaux de parallélisme correspondent :

- Vector : thread (qui fonctionnent en mode SIMD)
- Worker : ensemble de threads (blocks)
- Gang : ensemble de workers (Streaming Multiprocessor) qui partagent les ressources (mémoire partagée). Les gangs fonctionnent de manière indépendante.

Synthèse

- Courbe d'apprentissage trés rapide et demande peu d'investissement
- Les directives sont trés peu intrusives, ce qui facilite le débogage et permet des modifications incrémentales du code à paralléliser
- OpenACC peut être activé/désactivé au moment de la compilation sans avoir à modifier le code (les pragmas sont alors vus comme des commentaires)
- Le même code peut être compilé pour une architecture multicores CPU ou une architecture GPUs, etc...
- Contrairement aux directives OpenMP qui sont plus *prescriptives*, celles d'OpenACC sont plutôt *descriptives* : i.e le compilateur est libre de compiler le code de la façon qu'il estime la meilleure en fonction de l'architecture cible.

```
pgcc -ta=tesla:managed ...
gcc -fopenacc --target=nvptx-none ...
pgfortran -ta=multicore –fast -Minfo=acc ...
```

Glossaire

Glossaire

Bande passante: désigne le débit binaire maximal d'une voix de transmission. Elle représente la quantité d'informations (en GB/s) qui peut être transmise sur une voie de transmission.

Bus: dispositif de transmission de données partagé entre plusieurs composants d'un système numérique. ex: bus PCIe, Nvlink, etc...

CUDA: Compute Unified Device Architecture. API de programmation des GPGPU en C.

Double précision: Le format double précision est le plus couramment utilisé sur les ordinateurs actuels pour représenter les nombres en virgule flottante. Stocké sur 64 bits, il correspond à une mantisse de 53 bits, soit environ 16 chiffres décimaux, et permet de représenter des nombres de valeur absolue comprise entre $4,94 * 10^{-324}$ et $1,80 * 10^{308}$.

HBM2: [High Bandwidth Memory 2] nouvelle génération de la mémoire vidéo (GDDR5) qui possède un plus grande capacité (empilement de couches), une fréquence et donc une consommation réduite, et un large bus de données.

Glossaire

Latence: désigne le temps nécessaire à un paquet de données pour passer de la source à la destination à travers un réseau ou un bus de données.

Pipeline de rendu 3D: ensemble des étapes qui conduit à l'affichage d'une scène 3D à partir de données (maillages, textures).

Pipeline (pipelining): technique d'optimisation du nombre d'instructions par cycle.

Puissance crête: mesure les performances des unités de calcul en virgule flottante (FPU) contenues dans le cœur.

Rasterisation: la rastérisation, ou matricialisation, est un procédé qui consiste à convertir une image vectorielle en une image matricielle destinée à être affichée sur un écran ou imprimée par un matériel d'impression.

Kernel: Un kernel est une portion parallèle de code à exécuter sur le périphérique (GPU). Chacune de ses instances s'appelle un thread.

Merci de votre attention.

Références:

- https://www.saagie.com/fr/blog/l-histoire-du-deep-learning/
- https://perso.liris.cnrs.fr/christian.wolf/teaching/
- https://fr.wikipedia.org/wiki/Compute_Unified_Device_Architecture
- https://www.malekal.com/cpu-vs-gpu-les-differences/
- http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf
- https://www.science-et-vie.com/science-et-culture/1965-2020-la-loi-de-moore-est-morte-53613
- http://lyoncalcul.univ-lyon1.fr/events/2013/cours_archi.pdf
- https://www.labri.fr/perso/pbenard/teaching/pghp/slides/Cours_PGHP_2016_08_Cuda.pdf
- https://people.duke.edu/~ccc14/sta-663/GPUsAndCUDAC.html
- https://overclocking.com/comment-organise-un-gpu/
- https://overclocking.com/quelles-sont-les-unites-de-base-dun-gpu/
- https://www.malekal.com/cpu-vs-gpu-les-differences/
- https://moodle.insa-lyon.fr/mod/resource/view.php?id=38316
- http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf
- https://docs.computecanada.ca/wiki/OpenACC_Tutorial_-_Adding_directives
- https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9262-zero-to-gpu-hero-with-openacc.pdf

Programmation parallèle par échange de messages - MPI

Albanne Lecointre, Franck Pérignon

février-mars 2023

1 *Introduction*
- Quelques rappels sur le parallélisme
- Modèle de programmation parallèle par échange de messages
- Le standard MPI

2 *Premiers pas avec MPI*

3 *Communications*

4 *Types dérivés*

5 *Groupes, communicateurs et topologies*

6 *Autres fonctionnalités*

Objectifs de la programmation parallèle

Paralléliser?
"**Réorganiser**" le problème pour pouvoir traiter **simultanément** des données et des calculs, en utilisant plusieurs **ressources** de calcul.
Pourquoi?

- Améliorer les performances $\Rightarrow$ calculer plus vite.
- Traiter un volume de données plus important $\Rightarrow$ utiliser la mémoire de plusieurs noeuds de calcul.

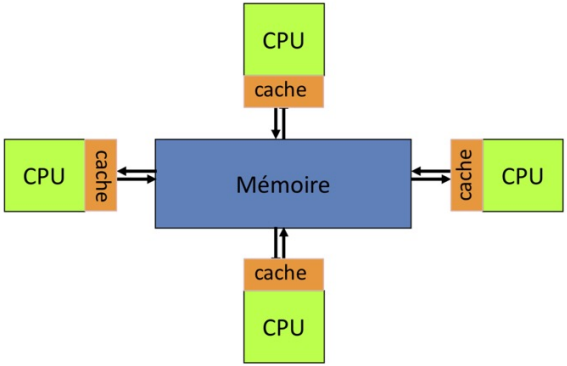
Points importants :

- connaître l'architecture matérielle : voir introduction de ce cours,
- choisir un modèle de programmation.

Architectures parallèles (1)

Calculateur à mémoire partagée

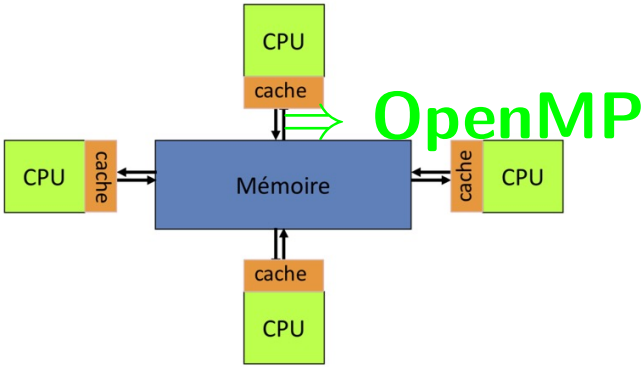
- Plusieurs processeurs partagent un même espace en **mémoire globale** via un bus mémoire rapide.



Architectures parallèles (1)

Calculateur à mémoire partagée

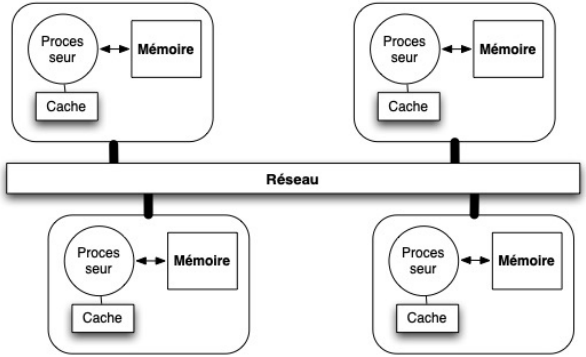
- Plusieurs processeurs partagent un même espace en **mémoire globale** via un bus mémoire rapide.



Architectures parallèles (2)

Calculateur à mémoire distribuée

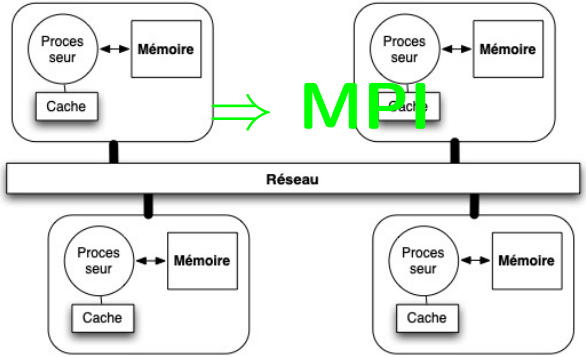
- Chaque nœud a sa propre **mémoire locale**.
- Chaque nœud accède à la mémoire des autres nœuds via le réseau (appel à des **routines de communications**).



Architectures parallèles (2)

Calculateur à mémoire distribuée

- Chaque nœud a sa propre **mémoire locale**.
- Chaque nœud accède à la mémoire des autres nœuds via le réseau (appel à des **routines de communications**).



Architectures parallèles (3)

Calculateur hybride

- Cas le plus courant en pratique : un ensemble de calculateurs à mémoire partagée et/ou distribuée (plus éventuellement des accélérateurs GPU ou autres) reliés par un réseau rapide.

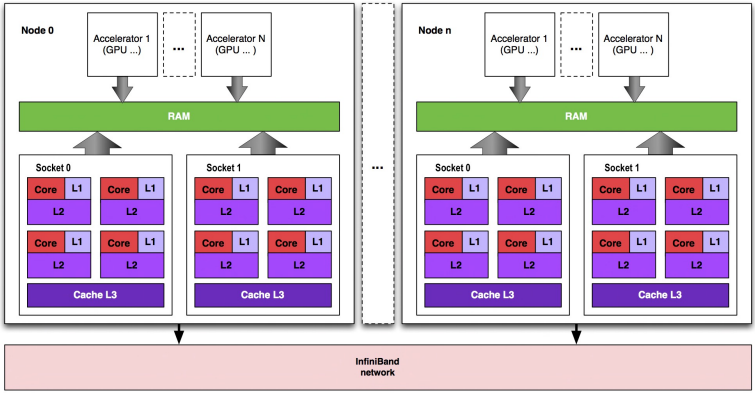
Exemple : le cluster "Dahu"

- plus de 100 noeuds pour 3600 coeurs de calcul
- 1 noeud fat (1TB RAM)
- 1 noeud de visualisation
- réseau OmniPath 100Gb/sec
- scratch rapide BeeGFS (bettik) partagé avec le cluster Luke

Evolution régulière, voir *https://gricad-doc.univ-grenoble-alpes.fr/hpc/description/* ou commande "recap.py" depuis la frontale pour les dernières mises à jour.

Quelques rappels de vocabulaire:

- **Cluster** : un ensemble de ressources (processeurs, mémoires, disques ...), distribuées sur plusieurs machines, connectées par un réseau performant.
- **Noeud** : une machine, un composant du cluster.
- **Socket** : processeur, circuit imprimé, deux coeurs ou plus.
- **Coeur** : une unité de calcul avec sa propre mémoire (cache, registers).



⚠ bien distinguer “coeur” physique et “processus” MPI, i.e. une instance d’exécution d’un programme (voir plus loin).

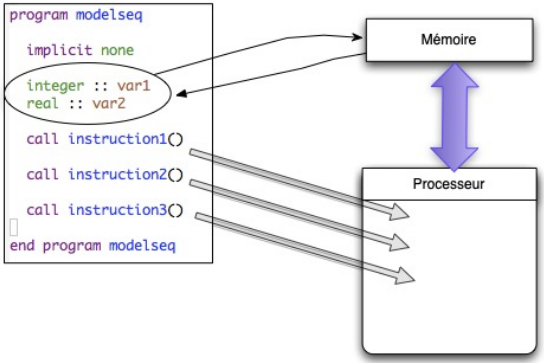
Ce qu'il faut retenir ...

En résumé, du point de vue du programmeur, il faut voir l'architecture comme un réseau de processeurs.

- Des unités de calcul ⇒ la puissance de calcul.
  Détermine le nombre d'opérations possibles par unité de temps (FLOPS/s).
- Différents niveaux de mémoire ⇒ différents niveaux de parallélisme.
  Critères déterminants : taille des mémoires et vitesse de lecture/écriture.
- Un réseau de communication ⇒ facteur limitant : bande passante, i.e. la quantité de données par unité de temps pouvant transiter d'une mémoire à l'autre.

Programmation parallèle : utilisation d'une couche logicielle pour gérer et optimiser l'utilisation et l'accès à ces ressources ⇒ **MPI**.

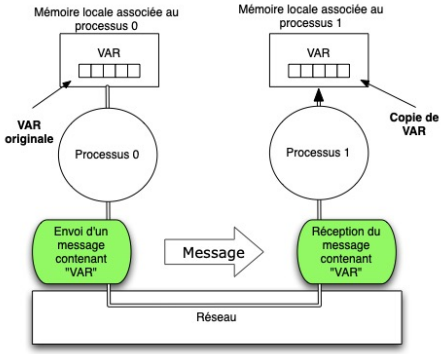
Modèle séquentiel



- Une seule unité de calcul exécute le programme et traite les instructions séquentiellement.
- Toutes les variables sont allouées dans la même zone mémoire.

Modèle de programmation par échange de messages

Principe : un ensemble de processus travaillant chacun sur des données locales et utilisant un protocole d'échange de messages pour partager leurs données.



Processus : une unité de calcul virtuelle, ≈ un programme placé sur un coeur physique.

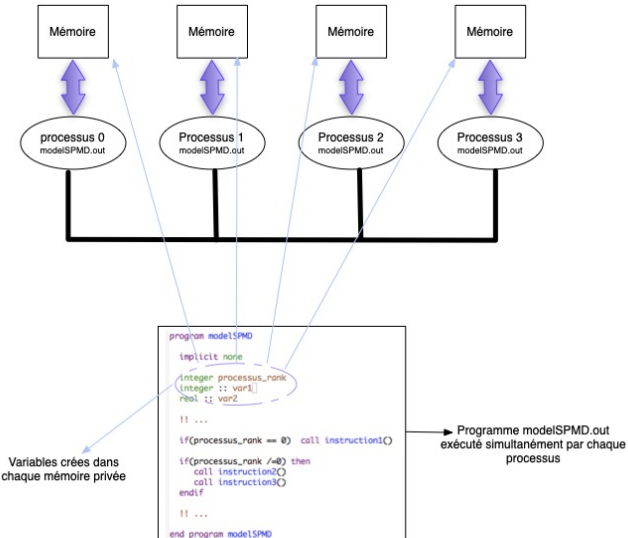
Message :

- données (i.e. contenu de la variable)
- description du message (enveloppe)

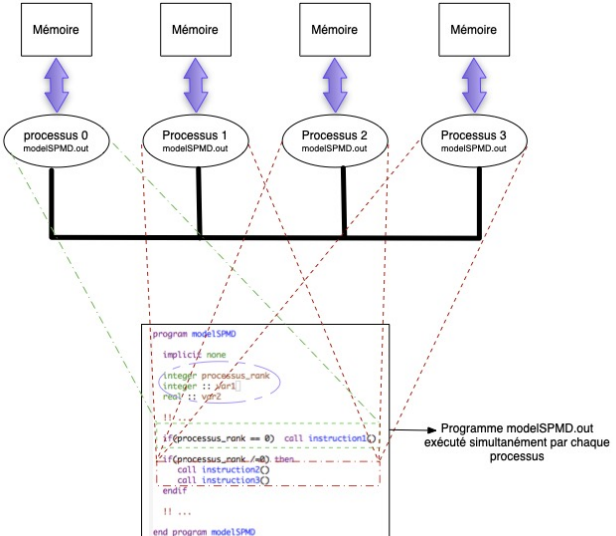
Notions clés: processus, message, synchronisation.

- Processus “virtuel” $\neq$ processeur/coeur physique. Le nombre de coeurs et le nombre de processus sont a priori différents et il peut donc y avoir plusieurs processus sur un même coeur.
- Chaque processus a sa propre mémoire, ses variables, et il n’a pas accès directement aux variables des autres processus.
- Le partage des données entre processus se fait par envois et réceptions explicites de messages.
- Synchronisation possible des processus.

Modèle SPMD : Single Program Multiple Data



Modèle SPMD : Single Program Multiple Data



Modèle SPMD

- SPMD, modèle le plus courant : chaque processus fait tourner le même exécutable sur des données différentes.
- Autre modèle : Multiple Program, Multiple Data (MPMD). Chaque processus fait tourner un programme différent sur des données différentes.

Modèle par échange de messages : avantages et inconvénients

(relativement à OpenMP ...)

Avantages

- Peut être implémenté sur une grande variété de plates-formes :
  - calculateur à mémoire distribuée,
  - calculateur à mémoire partagée,
  - architecture parallèle hétérogène.
- Permet en général un plus grand contrôle de la localisation des données et par conséquent une meilleure performance des accès aux données.
- Passage à l'échelle pour des volumes de données importants.

Inconvénients

- Nécessite de repenser et réécrire le code existant. Mise en oeuvre potentiellement plus compliquée qu'avec OpenMP.
- Le coût des communications peut devenir important.
- Attention à la répartition des tâches.

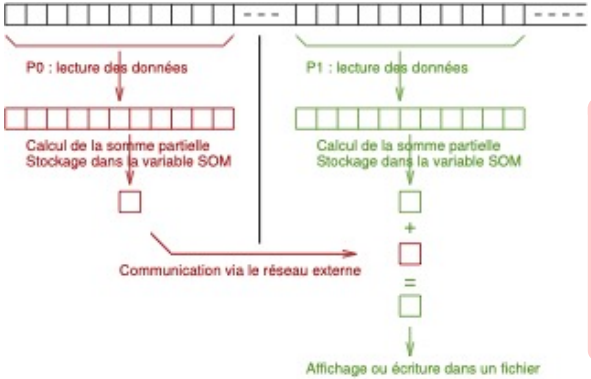
Comment paralléliser?

Que faire pour passer d'un code séquentiel à un code parallèle avec un modèle par échange de message?

- Identifier un type de parallélisme adapté au problème à traiter :
  - "data parallelism" : décomposition des données. Les données peuvent être découpées et distribuées sur différentes zones mémoires.
  - "task parallelism" : répartition des tâches sur les différents processus.
- Revoir la structure de donnée.
- Revoir/modifier l'ordre des calculs.
- Identifier les tâches indépendantes.
- Gérer les communications.
- ...

Un premier exemple simple de programme sur architecture à mémoire distribuée

Calcul de la somme des termes d'un vecteur.



Remarque: La parallélisation MPI et le nombre de processus mis en oeuvre peuvent modifier l'ordre d'exécution des opérations: ici le découpage en sommes partielles influence l'ordre selon lequel sont sommés les éléments du vecteur global.

Qu'est ce que MPI?

M essage P assing I nterface - Description et spécifications pour une implémentation du modèle par échange de message.

- Interface définie par une norme :
  MPI-1 (1994), MPI-2.1 (2008), MPI-2.2 (2009), MPI-3.1 (2015), MPI-4.0 (2021)
  Voir *https://www.mpi-forum.org/docs/mpi-4.0/*.
- Différentes implémentations/bibliothèques :
  - open source (openmpi, mpich),
  - propriétaires et/ou spécifiques à une machine particulière (i.e. optimisées par le constructeur).

  $\Rightarrow$ Plus ou moins complet selon l'implémentation.
- Utilisable dans plusieurs langages : Fortran, C, C++, Python.

En pratique : MPI est une interface portable mais les fonctionnalités et performances peuvent varier d'une implémentation à l'autre, en fonction de la machine, du compilateur, du langage utilisé ...

Ce qu'il faut retenir ...

Pour conclure cette première partie, quelques points clés à retenir

- MPI : programmation par échange explicite de messages.
- Message = données (contenu de la mémoire) + description des données (taille, type etc).
- MPI : un programme entièrement exécuté par tous les processus : tout est parallèle !
  $\Rightarrow$ différent d'OpenMP où alternent zones parallèles et zones séquentielles.
- Programmation intrusive : il faut entièrement repenser le code séquentiel existant.

Bibliographie

- Message Passing Interface Forum, http://www.mpi-forum.org/index.html
- "MPI: A Message-Passing Interface Standard, Version 4.0", Message Passing Interface Forum, june 2021 - http://mpi-forum.org/docs/
- Gropp, W., E. Lusk, and A. Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message Passing*. 2nd Edition. MIT Press, Cambridge, MA.
- **Cours et tutoriaux:**
  http://www.idris.fr/formations/mpi (Supports de cours et Aide mémoire en C et Fortran)
  http://www.mcs.anl.gov/research/projects/mpi/
- Quelques exemples de bibliothèques scientifiques parallèles :
  ScaLAPACK, PETSc, FFTW ...

1 *Introduction*

2 *Premiers pas avec MPI*
- Écriture d'un programme MPI
- Compilation et exécution d'un programme MPI
- Présentation des tps et consignes/aide-mémoire

3 *Communications*

4 *Types dérivés*

5 *Groupes, communicateurs et topologies*

6 *Autres fonctionnalités*

MPI

Des processus indépendants qui travaillent en parallèle et discutent entre eux en échangeant des messages.
Pour “utiliser” MPI, il faudra donc

- **Écrire un programme** qui devra (en plus des opérations habituelles)
  - créer et organiser un contexte de communication,
  - organiser les processus et répartir les tâches sur ceux-ci,
  - gérer les échanges de messages.
- **Exécuter** ce programme sur plusieurs processus : créer des processus, les répartir sur les coeurs physiques des machines de calcul et faire exécuter le programme sur chacun de ces processus.

Objectifs de cette première partie

- Apprendre à écrire un programme “minimal” MPI.
- Aperçu de la structure standard de ce type de programme.
- Apprendre à exécuter ce programme dans différents contextes.

Structure standard d'un programme MPI

Une bibliothèque MPI va fournir un certain nombre de fonctions pour chacun des langages disponibles. Quel que soit le langage, la structure standard du programme ressemblera en général à :

1. inclusion des headers, modules, packages ...
2. initialisation du contexte MPI
3. organisation des processus
4. calculs locaux et échanges de messages entre les processus
5. fin du contexte de communication.

La bibliothèque MPI

Les routines que nous allons aborder dans ce cours peuvent être classées selon les catégories ci-dessous :

- **Gestion du contexte de communication et des processus**
  - *Initialisation* et fin du contexte de communication. *Séance 1. TP 1.*
  - Organisation des processus en *communicateurs*, *groupes* ou *topologies*. *Séance 1 et 4 - TP 4.*
- **Communications point à point**.
  Échange de messages entre deux processus. *Séance 1 et 2. TP 2.*
- **Communications collectives.**
  Échange de messages entre des groupes de processus. *Séance 2. TP 2.*
- **Création/définition de nouveaux types de données :**
  Types dérivés MPI. *Séance 3, TP 3.*

Comment avoir accès aux fonctions/outils MPI dans vos programmes ?

En fortran : appel du module mpi_f08

```
use mpi_f08
```

Autres possibilités (⚠️ deprecated, à éviter !) : use mpi ou include 'mpif.h'.
En C/C++

```
#include <mpi.h>
```

En python

```
from mpi4py import MPI
```

Initialisation et clôture du contexte MPI

La première instruction appelée par MPI doit être celle qui "démarre" le contexte de communication :

```
int error = MPI_Init(&argc, &argv);
```

De même, à la fin du programme, chaque processus doit clore le contexte MPI :

```
error = MPI_Finalize();
```

Les deux routines renvoient un code d'erreur qui vaut *MPI_SUCCESS* si tout s'est bien passé.

- Un et un seul appel à *MPI_Init* par programme.
- Init impératif avant tout appel d'une autre fonction MPI.
- init/finalize: collectif, doit être appelé par tous les processus.

Un petit intermède à propos du langage ...

Les exemples donnés dans le cours seront écrits alternativement en C ou en Fortran.
Cependant, la convention de nommage pour les fonctions est la même, quelle que soit le langage :

```
MPI_Xxx_xxx ! Attention à respecter la casse en C/C++ !
```

Vous pourrez donc facilement interpréter les exemples et passer d'un langage à l'autre,

- il suffira d'adapter les types (int → integer ...),
- et de vérifier le nombre d'arguments C/Fortran qui est la plupart du temps presque identique, à un argument près : le dernier argument du Fortran est transformé en argument de retour pour le C

```
call MPI_Finalize(code_erreur) !! Fortran
```

```
int code_erreur = MPI_Finalize(); // C ou C++
```

👍Dans tous les cas nous vous recommandons de toujours vérifier dans la norme ou via man le type et le nombre d'arguments des fonctions MPI. Cela évitera des messages d'erreur pas toujours explicites ...

MPI et le C++

⚠ l'interface C++ est obsolète depuis MPI-2 et a été supprimée à partir de MPI-3.
Il est donc fortement déconseillé de l'utiliser.
Conseil :

- codez en C++ mais pour tous les appels MPI, utilisez l'API C,
- continuez d'utiliser un compilateur C++ (souvent plus "rigoureux" que le compilateur C).

Vous pouvez aussi éventuellement utiliser une implémentation spécifique telle que boost::mpi,
http://www.boost.org/doc/libs/1_62_0/doc/html/mpi.html.

Comment retrouver la signature des fonctions MPI ?

Une solution simple pour obtenir rapidement la signature des fonctions dans les différents langages : man function_name, qui vous donnera la signature pour le C et le fortran (oubliez le C++ !).

```
> man MPI_Comm_size
NAME
       MPI_Comm_size  - Returns the size of the group associated with a communicator.
SYNTAX
C Syntax
       #include <mpi.h>
       int MPI_Comm_size(MPI_Comm comm, int *size)

Fortran 2008 Syntax
       USE mpi_f08
       MPI_Comm_size(comm, size, ierror)
       TYPE(MPI_Comm), INTENT(IN) :: comm
       ...
```

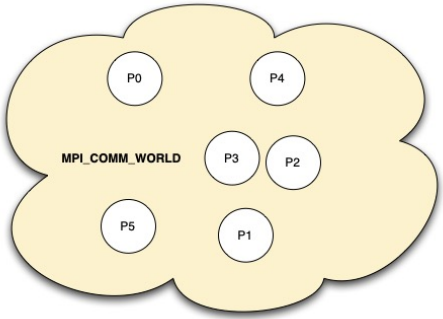
La référence restant le document *http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf*.
Pour python, voir *http://mpi4py.readthedocs.io/en/stable/* ou utiliser la documentation interactive.

Organisation des processus

Processus: une unité de calcul "virtuelle", placée sur un coeur physique.
Communicateur: ensemble de processus susceptibles de communiquer entre eux.

Un communicateur est caractérisé par

- son nom,
- sa **taille** : le nombre de processus qu'il contient, (accès : *MPI_Comm_size*),
- les **rangs** de ses processus : un identifiant pour chaque processus du communicateur, (accès : *MPI_Comm_rank*).

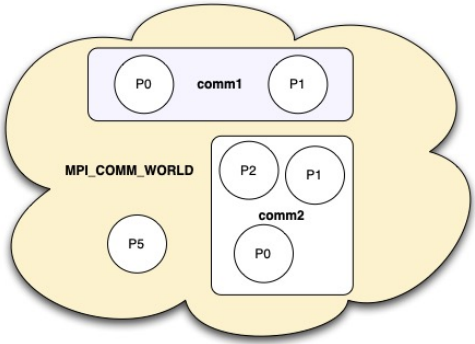


Organisation des processus

Processus: une unité de calcul "virtuelle", placée sur un coeur physique.

Communicateur: ensemble de processus susceptibles de communiquer entre eux.

- Plusieurs communicateurs peuvent coexister dans le même programme.
- Un processus peut appartenir à plusieurs communicateurs (et aura un rang différent dans chacun !).



Organisation des processus

- **MPI_COMM_WORLD** est le communicateur par défaut constitué de tous les processus attribués à l'application. Il est créé lors de l'initialisation du contexte de communication.
- ⚠ les processus ne peuvent communiquer que s'ils partagent un même communicateur.
  ⇒*Toutes les routines MPI auront un communicateur comme argument.*

La création de communicateurs permettra entre autres de :

- réduire/optimiser la portée des communications à un sous-ensemble de processus,
- classer et d'organiser les processus (par exemple sur des grilles, à l'aide des topologies).

Un premier exemple en Fortran

```
program testmpi
! 1 - Inclusion du module mpi
use mpi_f08
implicit none
type(MPI_Comm) :: mon_comm
integer :: error, nombre_de_processus, rang
! 2 - Initialisation du contexte de communication
call MPI_Init(error)
! 3 - Organisation des processus : par défaut dans MPI_COMM_WORLD (crée à l'appel de MPI_Init)
mon_comm = MPI_COMM_WORLD
! nombre_de_processus = taille du communicateur mon_comm
call MPI_Comm_size(mon_comm,nombre_de_processus,error)
! rang = rang du processus courant dans mon_comm
call MPI_Comm_rank(mon_comm, rang, error)
! 4 - Calculs, communications ...
print *, "je suis le processus de rang ", rang
! 5 - Cloture du contexte MPI
call MPI_Finalize(error)
end program
```

Un premier exemple en C

```c
#include <stdio.h>
// 1 - Inclusion du header mpi
#include <mpi.h>
int main(int argc, char* argv[])
{
  int error, nombre_de_processus, rang;
  // 2 - Initialisation du contexte de communication
  error = MPI_Init(&argc,&argv);
  // 3 - Organisation des processus : par défaut dans MPI_COMM_WORLD (crée à l'appel de MPI_I
  MPI_Comm mon_comm = MPI_COMM_WORLD;
  // nombre_de_processus = taille du communicateur mon_comm
  error = MPI_Comm_size(mon_comm,&nombre_de_processus);
  // rang = rang du processus courant dans mon_comm
  error = MPI_Comm_rank(mon_comm,&rang);
  // 4 - Calculs, communications ...
  printf("je suis le processus de rang %d\n",rang);
  // 5 - Cloture du contexte MPI
  error = MPI_Finalize();
}
```

Un premier exemple en Python

```python
# Inclusion des packages pour mpi
from mpi4py import MPI

# Pas d'initialisation explicite du contexte --> fait à l'import

mon_comm = MPI.COMM_WORLD
# Taille du communicateur
nombre_de_processus = mon_comm.Get_size()  # ou mon_comm.size

# Rang du processus courant dans mon_comm
rang = mon_comm.Get_rank()  # ou mon_comm.rank

print(f'je suis le processus de rang {rang}')

# Pas de cloture explicite du contexte
```

Exemple d'exécution

A l'exécution, avec 4 processus, on obtient :

```
je suis le processus de rang 0
je suis le processus de rang 2
je suis le processus de rang 1
je suis le processus de rang 3
```

- Le programme est exécuté simultanément sur chaque processus,
- ⚠ pour les sorties "écran", pas de règle pour l'ordre d'affichage.

Remarque : nous reviendrons plus loin sur comment compiler/linker puis exécuter un programme MPI.

Mesure du temps dans un programme MPI

Pour mesurer le temps écoulé sur chaque processus, on utilise la fonction *MPI_Wtime* : elle renvoie le temps écoulé depuis un temps arbitraire (constant pendant l'exécution du programme).

```
double starttime, elapsedtime;
starttime = MPI_Wtime();
//... calculs ...
elapsedtime   = MPI_Wtime() - starttime;
```

La résolution de cette fonction est donnée par *MPI_Wtick*.

Quelques routines utiles

Informations sur le système, l'implémentation, ...

```
// Obtenir le nom du processeur sur
// lequel tourne le processus courant
MPI_Get_processor_name(...)

// Obtenir la version de MPI
MPI_Get_version(...)
```

Vérification du contexte MPI, arrêt ...

```
// Test : vrai si MPI_Init a bien été appelé
MPI_Initialized (...)

// Tentative d'arret de tous les processus du communicateur
MPI_Abort(...)
```

Compilation/édition de lien

La compilation/édition de lien d'un programme MPI est similaire à celle d'un programme classique.

- On utilise les compilateurs "habituels" (voir le module outils de développement),
- avec des options spécifiques à MPI et des librairies à linker

Exemple (compilateur intel, fortran, implémentation MPI "par défaut"):

```
ifort -c monProgramme.f90
ifort -lmpi -lmpifort -lmpigi monProgramme.o -o ./monProg
```

ou

```
> g++ -I/nix/store/slfy56hw7zl8ff1kzq984vb15m1wj3bi-openmpi-3.1.0/include -pthread -Wl,-rp
-Wl,/nix/store/slfy56hw7zl8ff1kzq984vb15m1wj3bi-openmpi-3.1.0/lib -Wl,--enable-new-dtags
-L/nix/store/slfy56hw7zl8ff1kzq984vb15m1wj3bi-openmpi-3.1.0/lib -lmpi ./monProg
```

😟 les options, les chemins vers les librairies et les librairies elles mêmes vont dépendre de l'implémentation de MPI et du compilateur ...
C'est un peu pénible mais heureusement, certaines implémentations (e.g. openmpi) fournissent des "wrappers" pour la compilation et l'édition de lien.

Compilation/édition de lien - Wrappers mpi

- mpif90 (fortran), mpicc (C), mpic++ (C++),
- mpiifort (fortran, Intel), mpiicc (C Intel), mpiicpc (C++ Intel).

En général chaque wrapper a une option "show" permettant d'afficher les commandes réellement appelées, e.g :

```
# Exemple sur macos, avec openmpi
> mpic++ --showme
clang++ -I/usr/local/Cellar/open-mpi/4.0.0/include -L/usr/local/opt/libevent/lib -L/usr/lo

# Intel
> mpiicpc -show
...
```

⇒ l'appel aux wrappers évite d'avoir à connaître la ligne de compilation/link complète. Utilisez les !

Exécution d'un programme MPI

Exécuter un programme MPI signifie "distribuer" un exécutable sur plusieurs coeurs et faire tourner cet exécutable sur chacun d'eux, gérer les communications, la mémoire etc.
Il faudra donc transmettre un certain nombre d'informations à l'exécution, telles que

- le nombre de processus attribués à l'application,
- les noms des noeuds à utiliser,
- ...

Un script est fourni dans l'implémentation MPI : mpirun ou mpiexec.

```
mpirun <args> your_program
man mpirun # <args> va dépendre de l'implémentation et du système.
```

Exemples

```
# 4 processus, sur un seul noeud
mpirun -np 4 ./monProg
```

```
# calcul sur deux noeuds, 4 processus par noeud
mpirun -H luke44,luke43 -npernode 4 ./monProg
```

Nous verrons plus loin un exemple d'appel complet, via oar.

Quelques remarques à propos du choix de l'implémentation MPI

- Pour démarrer, utilisez openmpi basé sur les compilateurs 'gnu'', souvent disponibles par défaut.
- En général d'autres implémentations seront accessibles sur les machines de calcul (via nix ou guix par exemple)
- ⚠ à la compatibilité entre le compilateur utilisé et la commande mpirun. Assurez-vous (which mpirun) qu'il s'agit bien du mpirun correspondant aux compilateurs employés pour votre code.
- Quelques commandes utiles pour suivre le job et l'utilisation des ressources :
  top, *htop*, *oarstat* (si oar est le gestionnaire de ressource) ...

Résumé et exemples (1)

1. Utilisez nix ou guix.
2. Vérifiez l'implémentation, ce que font les wrappers et le compilateur associé.

Exemple, compilateurs gnu:

```
>> switch ced_gnu
>> nix-env -i openmpi
>> mpirun --version
mpirun (Open MPI) 3.1.0
>> which mpif90
/home/perignon/.nix-profile/bin/mpif90
>> which mpirun
/home/perignon/.nix-profile/bin/mpirun
>> mpif90 --showme
gfortran -I/nix/store/slfy56hw7zl8ff1kzq984vb15m1wj3bi-openmpi-3.1.0/include -pthread -I/nix
>> which gfortran
/home/perignon/.nix-profile/bin/gfortran
```

Exemple, utilisation des compilateurs intel sur Dahu

```
>> switch ced_intel
>> nix-env -i -A nur.repos.gricad.intel-oneapi
>> source ~/.nix-profile/setvars.sh

>> mpirun --version
Intel(R) MPI Library for Linux* OS, Version 2021.5 Build 20211102 (id: 9279b7d62)
Copyright 2003-2021, Intel Corporation.

>> mpiifort -show
ifort -I"/nix/store/a5ha9bvp40s61nxsrybb9z5sv071689k-intel-oneapi-2022.1.2.146/mpi/2021.5.1//i
```

Compléments à propos de l'utilisation de mpirun

- Pour envoyer une variable d'environnement à chaque processus :

```
mpirun -x LD_LIBRARY_PATH -np 4 ./monProg
```

- Pour préciser l'emplacement de l'implémentation MPI :

```
mpirun --prefix /opt/MPI/install-gnu/openmpi-1.6.1-gnu-4.7.1 -np 4  ./monProg
```

- Fournir un fichier contenant les ressources (noeuds/coeurs) sur lesquelles le programme doit s'exécuter

```
mpirun --machinefile nomfichier -np 2  ./monProg
```

- Un peu d'aide ?

```
man mpirun
man mpiexec
```

Compléments à propos de l'utilisation de mpirun

Pour lancer votre programme vous devrez utiliser oar en mode interactif ou en batch.
Nous vous fournirons des scripts et des détails durant les tps.

- oarscript-gnu.sh pour le mode batch, basé sur les compilateurs gnu et openmpi
- oarscript-intel.sh pour le mode batch, basé sur les compilateurs intel sur Dahu.

Trainings - MPI, les bases

Comme pour OpenMP, les exercices sont décrits dans le projet gitlab ced2022/trainings.
Avant de passer aux communications, lisez les consignes et faites le premier exercice, “Bases de MPI”.

1 *Introduction*

2 *Premiers pas avec MPI*

3 *Communications*
- Communications point à point
- Optimisations des communications point à point
- Communications collectives

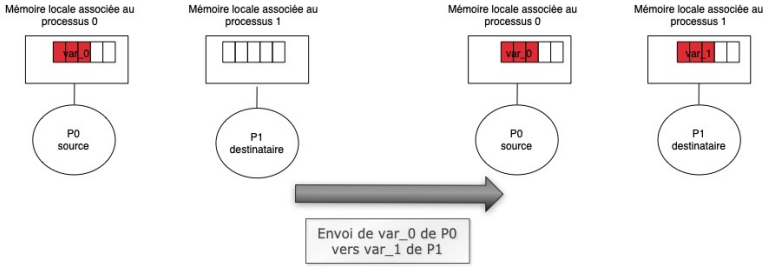
4 *Types dérivés*

5 *Groupes, communicateurs et topologies*

6 *Autres fonctionnalités*

Communication point à point : principe

envoi d'un *message* d'un processus *source* vers un processus *destinataire*.



- Finalité : copier une zone mémoire du processus source vers une zone mémoire du processus destinataire.
- Source : envoi explicite d'un message →MPI_Send.
- Destinataire : réception explicite d'un message → MPI_Recv.

Message?

- Une enveloppe : description du “contexte”.
  - Identifiants des processus source et destinataire (leurs rangs respectifs),
  - communicateur contenant la source et le destinataire,
  - tag : identifiant du message.
- Un corps de message : les données proprement dites.
  - Un “buffer” contenant les données,
  - le type des données,
  - leur taille.

Envoi et réception de message

Envoi d'un message Corps du message : les données — Enveloppe : le contexte

```
MPI_Send(data, nb_elements, type_data, destinataire, tag, communicateur)
```

Réception d'un message

```
MPI_Recv(data, nb_elements, type_data, source, tag, communicateur, status)
```

"nb_elements" : nombre maximum d'éléments de type "type_data" qu'on pourra recevoir dans data.
⚠ vous pouvez choisir un nombre différent du nombre effectifs d'éléments reçus (pas de vérification à la réception. Pour obtenir le nombre d'élements réellement reçus, voir la fonction *MPI_Get_count*).
👍vérifiez toujours dans la norme ou via man le type et le nombre d'arguments des fonctions MPI. Cela évitera des messages d'erreur pas toujours explicites ...

Un exemple simple

```
integer :: source, dest, tag, error, buffer, nb_elements
TYPE(MPI_Status) :: status
source = 0 ;  dest = 1; tag = 21 ; nb_elements=1
!! On récupère le rang du processus courant
call MPI_Comm_rank(MPI_COMM_WORLD, rang_processus, error)
```

Le processus “source” envoie un message

```
if(rang_processus == source) then
  buffer = 12      ! ici, buffer indique une zone de la mémoire du proc. 0
  call MPI_Send(buffer, nb_elements, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, error)
  print *, "Processus de rang :", rang_processus, ", buffer =", buffer
```

Le processus “dest” reçoit un message

```
else if(rang_processus == dest) then
  buffer = 0 ! ici, buffer indique une zone de la mémoire du proc. 1, qui
  ! n'a rien à voir avec "buffer" du proc 0.
  print *, "Processus de rang", rang_processus, ", buffer =", buffer
  call MPI_Recv(buffer, nb_elements, MPI_INTEGER, source, tag, MPI_COMM_WORLD, status, error)
  print *, "Processus de rang", rang_processus, ", nouveau buffer =", buffer
end if
```

Un exemple simple

```
>> Processus de rang 1, buffer =0
>> Processus de rang 0, buffer =12
>> Processus de rang 1, nouveau buffer =12
```

- Chaque processus exécute le programme.
- Les variables sont locales à chaque processus.
- L'envoi et la réception sont explicites.
- A chaque envoi/réception doit correspondre une réception/envoi .

Compléments

Wildcards : constantes pré-définies, évite d'avoir à renseigner le tag et/ou la source; permet d'autoriser la réception de n'importe quelle source et/ou avec n'importe quel tag.

```
source = MPI_ANY_SOURCE
tag = MPI_ANY_TAG
call MPI_Recv(buffer,12,MPI_INTEGER,source,tag,MPI_COMM_WORLD,status,error)
```

MPI_Recv : ⚠ à ne pas oublier l'argument '**status**'. C'est une structure qui contient les informations concernant l'enveloppe du message reçu :

- status%MPI_SOURCE : origine du message (rang)
- status%MPI_TAG : tag du message
  Exemple d'utilisation :

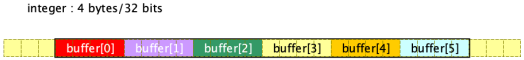
  ```
  call MPI_Get_count(status, MPI_INTEGER, count, erreur)
  ```

  count : nombre d'éléments reçus.
- L'argument **status** peut-être ignoré et remplacé par *MPI_STATUS_IGNORE*.

Echange de message : les types MPI

Pour tous les envois/réceptions de message il est impératif de préciser le type des données échangées.
"Type" MPI $\neq$ type variables C/C++ fortran
"Type" MPI $\approx$ comment "retrouver" dans la mémoire les données à transférer

```
int buffer[6];
int nombre = 3;
// cas 1 : envoi de trois entiers
MPI_Send(buffer, nombre, MPI_INT, ...);
// cas 2 : aucune erreur de compilation
// mais mauvais résultat
MPI_Send(buffer, nombre, MPI_DOUBLE ...)
```

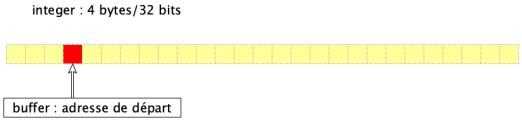


Ecriture d'un tableau d'entiers en mémoire

Echange de message : les types MPI

Pour tous les envois/réceptions de message il est impératif de préciser le type des données échangées.
"Type" MPI $\neq$ type variables C/C++ fortran
"Type" MPI $\approx$ comment "retrouver" dans la mémoire les données à transférer

```
int buffer[6];
int nombre = 3;
// cas 1 : envoi de trois entiers
MPI_Send(buffer, nombre, MPI_INT, ...);
// cas 2 : aucune erreur de compilation
// mais mauvais résultat
MPI_Send(buffer, nombre, MPI_DOUBLE ...)
```



Envoi de MPI_INT

Echange de message : les types MPI

Pour tous les envois/réceptions de message il est impératif de préciser le type des données échangées.
"Type" MPI $\neq$ type variables C/C++ fortran
"Type" MPI $\approx$ comment "retrouver" dans la mémoire les données à transférer

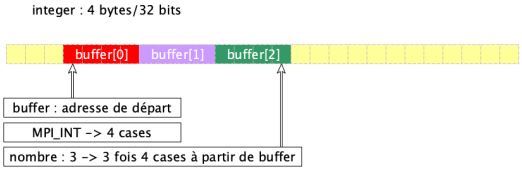
```
int buffer[6];
int nombre = 3;
// cas 1 : envoi de trois entiers
MPI_Send(buffer, nombre, MPI_INT, ...);
// cas 2 : aucune erreur de compilation
// mais mauvais résultat
MPI_Send(buffer, nombre, MPI_DOUBLE ...)
```

Envoi de MPI_INT

Echange de message : les types MPI

Pour tous les envois/réceptions de message il est impératif de préciser le type des données échangées.
"Type" MPI $\neq$ type variables C/C++ fortran
"Type" MPI $\approx$ comment "retrouver" dans la mémoire les données à transférer

```
int buffer[6];
int nombre = 3;
// cas 1 : envoi de trois entiers
MPI_Send(buffer, nombre, MPI_INT, ...);
// cas 2 : aucune erreur de compilation
// mais mauvais résultat
MPI_Send(buffer, nombre, MPI_DOUBLE ...)
```

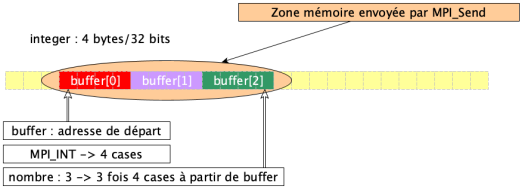


Envoi de MPI_INT

Echange de message : les types MPI

Pour tous les envois/réceptions de message il est impératif de préciser le type des données échangées.
"Type" MPI $\neq$ type variables C/C++ fortran
"Type" MPI $\approx$ comment "retrouver" dans la mémoire les données à transférer

```
int buffer[6];
int nombre = 3;
// cas 1 : envoi de trois entiers
MPI_Send(buffer, nombre, MPI_INT, ...);
// cas 2 : aucune erreur de compilation
// mais mauvais résultat
MPI_Send(buffer, nombre, MPI_DOUBLE ...)
```

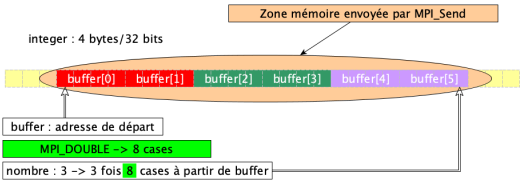


Envoi de MPI_INT

Echange de message : les types MPI

Pour tous les envois/réceptions de message il est impératif de préciser le type des données échangées.
"Type" MPI $\neq$ type variables C/C++ fortran
"Type" MPI $\approx$ comment "retrouver" dans la mémoire les données à transférer

```
int buffer[6];
int nombre = 3;
// cas 1 : envoi de trois entiers
MPI_Send(buffer, nombre, MPI_INT, ...);
// cas 2 : aucune erreur de compilation
// mais mauvais résultat
MPI_Send(buffer, nombre, MPI_DOUBLE ...)
```



Envoi de MPI_DOUBLE

⚠ Pas de vérification de consistance entre le type de la donnée envoyée et le type MPI !
⚠ send et recv sont indépendants : il est tout a fait possible de recevoir un envoi d'entiers dans des flottants.

Envoi de message : types des données échangées

Voici ci-dessous les types MPI pré-définis et leur correspondances avec les principaux types des langages C/Fortran.

Types Fortran	Types MPI	Types C	Types MPI
integer	MPI_INTEGER	signed in	MPI_INT
real	MPI_REAL	float	MPI_FLOAT
double precision	MPI_DOUBLE_PRECISION	double	MPI_DOUBLE
complex	MPI_COMPLEX	float _Complex	MPI_C_COMPLEX
complex double	MPI_DOUBLE_COMPLEX	double _Complex	MPI_C_DOUBLE_COMPLEX
logical	MPI_LOGICAL	_Bool	MPI_C_BOOL
character	MPI_CHARACTER	signed char	MPI_SIGNED_CHAR

	MPI Fortran	MPI C
Communicateur	Type(MPI_Comm)	MPI_Comm
Statut	Type(MPI_Status)	MPI_Status

Nous verrons plus loin qu'il est possible de définir ses propres types (types dérivés).

Comportement à l'exécution

Au cours d'un échange point à point, l'envoi et la réception (appels à MPI_Send et MPI_Recv) sont deux opérations distinctes éventuellement asynchrones, réalisées par deux processus différents.
Un certain nombre de questions se posent ...

- Que se passe t'il si aucune réception ne correspond à l'envoi?
- Peut-on utiliser les variables envoyées/reçues sans impact sur le message?
- Peut-on reprendre la main et faire autre chose pendant l'envoi ou la réception du message?
- ...

Deux notions importantes à définir : celle d'échange bloquant et celle de complétion d'un envoi ou d'une réception.

Complétion d'un message

- **Complétion de la réception** : le message est bien arrivé et la variable a été copiée dans la mémoire locale et est donc utilisable par le processus récepteur.
- **Complétion de l'envoi** : la variable envoyée (i.e. la zone mémoire correspondante) peut-être utilisée de manière sûre, en lecture ou en écriture, au sens ou une modification de cette variable par le processus émetteur n'aura plus d'impact sur la réception de l'autre processus.

$\Rightarrow$ complétion $\approx$ les variables envoyées/reçues sont utilisables sans risque.

Appel bloquant

Un appel est dit **bloquant** lorsque le programme reste dans la routine tant que l'envoi ou la réception n'est pas complet au sens défini juste avant.

⇒ un appel non-bloquant ne garantit pas la complétion.

Comportement à l'envoi

1. Démarrage de la communication (on parle de **postage** de l'envoi).
2. Ensuite, deux situations sont possibles.
   - **Bufferisation**, copie dans une zone mémoire temporaire en attendant l'envoi qui se fera plus tard, de manière asynchrone. Il n'est donc pas nécessaire d'attendre la réception.
   - **Synchronisation** avec la réception : la routine attend que le processus récepteur soit prêt et que le transfert ait commencé. Le programme reprend la main uniquement quand l'envoi est complet.

Comportement à l'envoi

1. Démarrage de la communication (on parle de **postage** de l'envoi).
2. Ensuite, deux situations sont possibles.
   - **Bufferisation**, copie dans une zone mémoire temporaire en attendant l'envoi qui se fera plus tard, de manière asynchrone. Il n'est donc pas nécessaire d'attendre la réception.
   - **Synchronisation** avec la réception : la routine attend que le processus récepteur soit prêt et que le transfert ait commencé. Le programme reprend la main uniquement quand l'envoi est complet.

La "bufferisation" donnera *probablement* de meilleures performances mais nécessite une copie en mémoire (côté émetteur ou récepteur, selon l'implémentation). A noter que la taille des buffers disponibles est forcément limitée.
Le passage d'un mode à l'autre **dépendra de l'implémentation et de la taille du message**.

Comportement à la réception

1. Initialisation de la réception, postage : vérification dans la pile des messages en attente si une enveloppe correspond à celle demandée par le *MPI_Recv*.
2. Transfert des données reçues dans la zone mémoire attribuée par le *MPI_Recv*.

Attention : seule l'enveloppe est vérifiée par le *MPI_Recv*, pas le type des données reçues!

Communications en mode standard

Pour conclure, dans le cas des communications en mode “standard” (i.e. avec MPI_Send et MPI_Recv):

- l'envoi de message peut-être “bufferisé” ou synchrone avec la réception,
- les envois et réceptions ont un comportement bloquant.

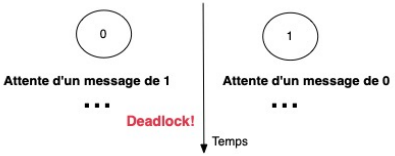
L'emploi de *MPI_Send* et *MPI_Recv* est donc sûr du point de vue de l'accès aux données mais une mauvaise gestion dans le code de la synchronisation et du comportement bloquant peut amener à des situations de **deadlock**.

Un premier exemple de deadlock

```
if( rang_du_processus == 0 )then
  ! Reception de 10 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
  ! Envoi de 10 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
else if (rang_du_processus == 1 )then
  !  Reception de 10 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
  ! Envoi de 10 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
end if
```

Un premier exemple de deadlock

```
if( rang_du_processus == 0 )then
  ! Reception de 10 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
  ! Envoi de 10 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
else if (rang_du_processus == 1 )then
  !  Reception de 10 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
  ! Envoi de 10 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
end if
```



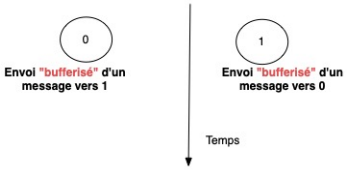
Les communications des lignes 3 et 8 (MPI_Recv bloquants) ne seront jamais complétées !

Un second exemple de deadlock

```
if( rang_du_processus == 0 )then
  ! Envoi de 10 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```

Un second exemple de deadlock

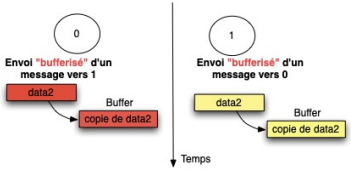
```
if( rang_du_processus == 0 )then
  ! Envoi de 10 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```



Cas d'un envoi bufferisé (petit message)

Un second exemple de deadlock

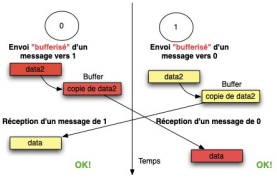
```
if( rang_du_processus == 0 )then
  ! Envoi de 10 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```



Cas d'un envoi bufferisé (petit message)

Un second exemple de deadlock

```
if( rang_du_processus == 0 )then
  ! Envoi de 10 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```



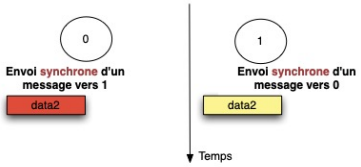
Cas d'un envoi bufferisé (petit message)

Un second exemple de deadlock

```
if( rang_du_processus == 0 )then
  ! Envoi de 10000 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10000, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10000 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10000, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10000 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10000, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10000 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10000, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```

Un second exemple de deadlock

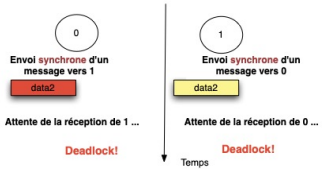
```
if( rang_du_processus == 0 )then
  ! Envoi de 10000 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10000, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10000 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10000, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10000 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10000, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10000 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10000, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```



Cas d'un envoi synchrone (par ex. augmentation de la taille du message)

Un second exemple de deadlock

```
if( rang_du_processus == 0 )then
  ! Envoi de 10000 MPI_REAL de data2 vers 1 avec le tag "tag2"
  call MPI_Send(data2, 10000, MPI_REAL, 1, tag2, MPI_COMM_WORLD, erreur)
  ! Reception de 10000 MPI_REAL dans data, en provenance de 1 avec le tag "tag1"
  call MPI_Recv(data, 10000, MPI_REAL, 1, tag1, MPI_COMM_WORLD, status, erreur)
else if (rang_du_processus == 1 )then
  ! Envoi de 10000 MPI_REAL de data2 vers 0 avec le tag "tag1"
  call MPI_Send(data2, 10000, MPI_REAL, 0, tag1, MPI_COMM_WORLD, erreur)
  ! Reception de 10000 MPI_REAL dans data, en provenance de 0 avec le tag "tag2"
  call MPI_Recv(data, 10000, MPI_REAL, 0, tag2, MPI_COMM_WORLD, status, erreur)
end if
```



Cas d'un envoi synchrone (par ex. augmentation de la taille du message)

Compléments - Envoi/Réception groupés

Il existe des routines permettant d'effectuer un envoi et réception en une seule instruction :

Message envoyé

```
// Un send et un recv dans la même commande
MPI_Sendrecv(data_send, nb_elem_send, type_send, destinataire,tag_send,
             data_recv, nb_elem_recv, type_recv, source, tag_recv, communicateur, status);
```

Message reçu

- ⚠ il n'y a pas nécessairement de correspondance entre le message reçu et le message envoyé.
- ⚠ *data_send* et *data_recv* doivent être différents.

données envoyées ET reçues

```
// Un seul buffer pour l'envoi et la réception
MPI_Sendrecv_replace(data, nb_elem, type, destinataire, tag_send, source, tag_recv,
                     communicateur, status);
```

Performances d'un calcul MPI

Quels sont les facteurs déterminants?

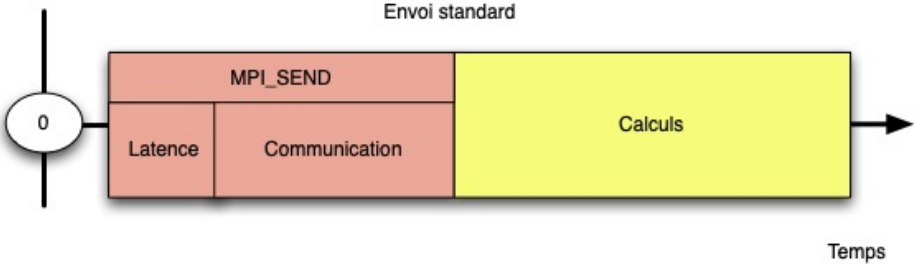
- L'architecture du système et le réseau entre les coeurs et les noeuds.
- L'implémentation de MPI utilisée.
- Le code que vous écrivez : choix des algorithmes, gestion de la mémoire, rapport communication/calcul dans le code, équilibrage des charges ...

Performances d'un calcul MPI

Partage du temps à l'exécution d'un programme MPI

- Latence : temps pour "démarrer" un échange $\approx$ le temps qu'il faut pour envoyer un message vide.
- Communications.
- Calculs.

Exemple :



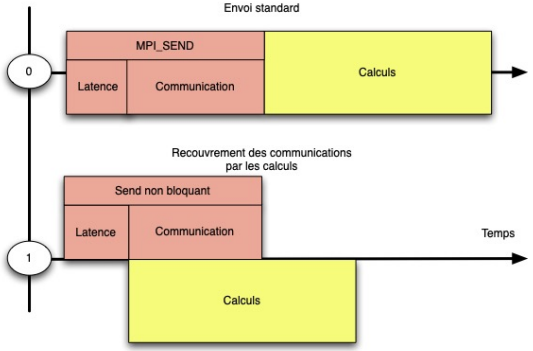
Comment améliorer l'implémentation?

- Utiliser les bons algorithmes ...
- Utiliser des librairies spécialisées (fftw, scalapack ...).
- Recouvrir les communications par des calculs.
- Changer le mode de communication.
- Équilibrer la charge entre les différents processus.

Comment améliorer l'implémentation?

- Utiliser les bons algorithmes ...
- Utiliser des librairies spécialisées (fftw, scalapack ...).
- Recouvrir les communications par des calculs.
- Changer le mode de communication.
- Équilibrer la charge entre les différents processus.

Exemple :



Communications non bloquantes

Comment? Ne pas attendre la complétion pour rendre la main.

Communications non bloquantes

Comment? Ne pas attendre la complétion pour rendre la main.

Envoi non-bloquant

MPI_Isend : dès que le message est posté, le programme reprend la main sur le processus source.

Réception non-bloquante

MPI_Irecv : dès que la réception est postée, le programme reprend la main.
⇒ le programme peut faire autre chose pendant que les transferts de message : recouvrement des communications par les calculs.
Attention: le programme reprend la main avant que la réception ou l'envoi soit complet. La variable envoyée/reçue ne sera donc pas utilisable immédiatement.

Exemples d'appels de communications non bloquantes

```
call MPI_Isend(data, nb_elements, type, destinataire, tag, communicateur, request, erreur)
```

```
call MPI_Irecv(data, nb_elements, type, source, tag, communicateur, request, erreur)
```

- A noter la présence d'un nouvel argument "requête" qui permettra d'identifier l'appel.

```
! en fortran
Type(MPI_Request) request
```

```
// en c
MPI_request request
```

- ⚠ pas d'argument "status" pour la réception non-bloquante. Les informations correspondantes sont accessibles via la requête.

Comment savoir si la réception ou l'envoi est terminé ?

En cas de communication non bloquante, il est nécessaire de vérifier quand celle-ci se termine. Deux méthodes pour cela : wait et test.

Test bloquant : MPI_Wait

```
call MPI_Isend(data, nb_elements, type, destinataire, tag, communicateur, request, erreur)
 !! ... Calculs ...
call MPI_Wait(request, statut, erreur)
```

MPI_Wait est bloquant et rend la main dès que la réception ou l'envoi est complet.

Comment savoir si la réception ou l'envoi est terminé ?

En cas de communication non bloquante, il est nécessaire de vérifier quand celle-ci se termine. Deux méthodes pour cela : wait et test.

Test bloquant : MPI_Wait

```
call MPI_Isend(data, nb_elements, type, destinataire, tag, communicateur, request, erreur)
 !! ... Calculs ...
call MPI_Wait(request, statut, erreur)
```

MPI_Wait est bloquant et rend la main dès que la réception ou l'envoi est complet.

Test non bloquant : MPI_Test

```
logical flag
call MPI_Isend(data, nb_elements, type, destinataire, tag, communicateur, request, erreur)
!! ... calcul ...
call MPI_Test(request, flag, statut, erreur)
```

MPI_Test est non-bloquant et renvoie un booléen qui est vrai quand l'envoi ou la réception est complet.

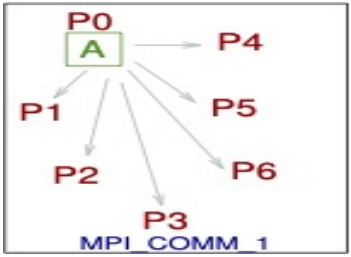
Modes de communication

Un autre moyen pour optimiser les communications consiste à **contrôler le mode d'envoi**.
Il existe quatre possibilités. Dans chaque cas l'envoi peut-être bloquant ou non-bloquant.

- **Mode standard** : celui que nous avons vu jusqu'à présent.
  MPI_Send, MPI_Isend.
- **Mode synchrone** : on force l'attente de la réception et du début du transfert.
  MPI_Ssend, MPI_Issend.
- **Mode bufferisé** : on force la copie des variables envoyées dans un buffer. C'est à l'utilisateur de gérer les buffers.
  MPI_Bsend, MPI_Ibsend.
- **Mode ready** : la réception doit avoir été postée avant l'envoi.
  A éviter ...
  MPI_Rsend, MPI_Irsend.

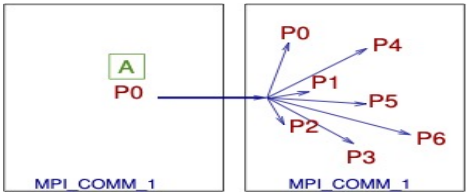
Communications collectives

- Rappel : communications point à point.
  - Communication entre 2 processus au sein d'un même communicateur.
  - Pour réaliser plusieurs envois successifs (d'une unique ou de différentes zones mémoire) : plusieurs appels à la même procédure MPI_Send (par exemple).
  - Certains inconvénients : lisibilité du code, latence à chaque appel, gestion des étiquettes (tag), ...



Communications collectives

Concept: communication entre un processus qui envoie (source) et un groupe de processus qui reçoit (destinataires).



- Procédures de communication qui s'effectuent sur un groupe de processus en une seule opération .
- Peuvent être appelées sur n'importe quel communicateur valide.
- Réalisent un ensemble de communications point à point avec un seul appel : optimisation.
- Exécuté par tous les processus du communicateur (P0 fait aussi partie du groupe de processus qui recoivent le message).

Communications collectives : caractérisation

- Une seule opération équivalente à une série de communications point à point.
- Une procédure spécifique fournie par la librairie et optimisée.
- Un seul et même appel pour tous les processus.
- Procédures de communication sur un ensemble de processus, identifié par un communicateur.
- La gestion des étiquettes est à la charge du système (non gérée par l'utilisateur).

Communications collectives

Les différentes procédures

- Synchronisation globale
  - MPI_Barrier()
- Echange de messages : transfert de données
  - Diffusion globale de données : MPI_Bcast()
  - Diffusion selective de données : MPI_Scatter()
  - Collecte selective de données réparties : MPI_Gather()
  - Collecte par tous les processus de données réparties : MPI_Allgather()
  - Collecte + Diffusion par tous les processus de données réparties ($\simeq$ "transposition") : MPI_Alltoall()
- Opérations sur les données lors des transferts
  - Réduction : collecte + opération sur des données : MPI_Reduce()
  - Réduction : collecte + opération sur des données + diffusion du résultat : MPI_Allreduce()
  - Opérations : addition, multiplication, min, max ou une opération créée par l'utilisateur

Remarque: Ces procédures de communications collectives existent aussi dans leur version non bloquante (MPI_Ixxx()), avec possibilité d'utiliser les appels de complétion sur la requête associée (MPI_Wait(), MPI_Test())

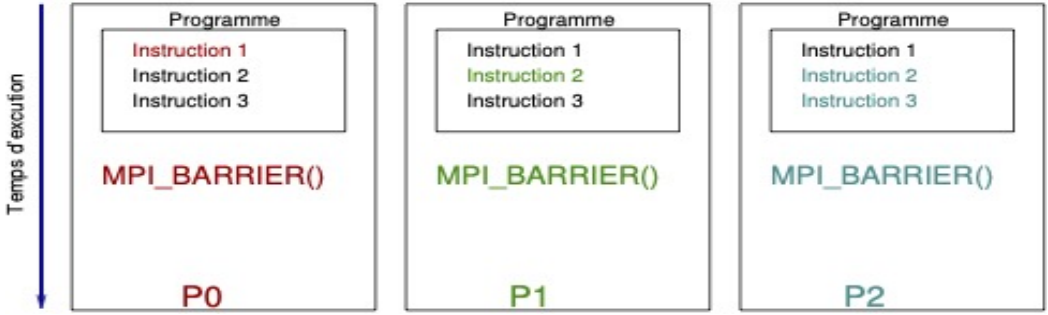
Communications collectives

Détails des procédures

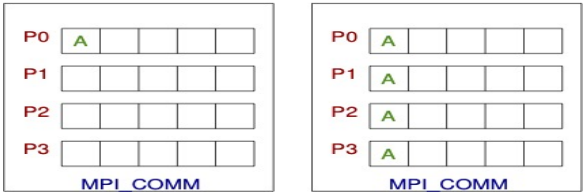
- Barrière de synchronisation : MPI_Barrier()

```
TYPE(MPI_Comm) :: comm
call MPI_Barrier(comm,ier)
```

- Attention: tout les processus communiquent avec tous les processus



Diffusion de données : MPI_Bcast




```
TYPE(MPI_Datatype) :: send_type
call MPI_Bcast(send_buffer, send_count, send_type, rank, comm, err)
```

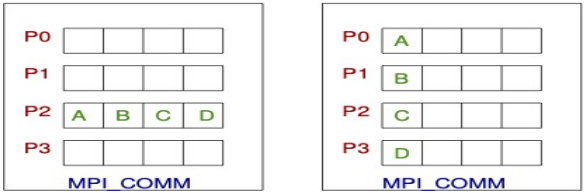
- send_buffer : adresse de début du buffer à envoyer
- send_count : nombre d'éléments à envoyer
- send_type : type MPI des données envoyées
- rank : processus qui envoie, les autres reçoivent
- comm : communicateur MPI
- err : code d'erreur

Diffusion de données : exemple d'utilisation

```
> mpirun -np 3 ./a.out | sort
P 0 avait 0 avant le broadcast depuis 2
P 0 possede 1002 après le broadcast depuis 2
P 1 avait 0 avant le broadcast depuis 2
P 1 possede 1002 après le broadcast depuis 2
P 2 avait 1002 avant le broadcast depuis 2
P 2 possede 1002 après le broadcast depuis 2
```

```
PROGRAM bcast
 implicit none
 use mpi_f08
 integer ::rang,ier,nb_procs
 integer :: valeur=0
 call MPI_Init(ier)
 call MPI_Comm_size(MPI_COMM_WORLD,nb_procs,ier)
 call MPI_Comm_rank(MPI_COMM_WORLD,rang,ier)
 if (rang == 2) valeur=rang+1000
 write(6,'(3(a,i0))')'P ',rang,' avait ',valeur,&
    ' avant le broadcast depuis 2'
 call MPI_Bcast(valeur,1,MPI_INTEGER,2,MPI_COMM_WORLD,ier)
 write(6,'(3(a,i0))')'P ',rang,' possede ',valeur,&
    ' après le broadcast depuis 2'
 call MPI_Finalize(ier)
END PROGRAM bcast
```

Diffusion sélective : MPI_Scatter




```
ier = MPI_Scatter(send_buffer, send_count, send_type,
                  recv_buffer, recv_count, recv_type, source, MPI_Comm);
```

- send_buffer / recv_buffer : adresse de début du buffer d'envoi / de réception
- send_count / recv_count : nombre d'éléments à envoyer/recevoir
- send_type / recv_type : type MPI de données envoyées/reçues
- source : rang du processus qui propage les données

Les couples (send_count,send_type) et (recv_count,recv_type) doivent assurer l'égalité entre les quantités de données envoyées et reçues.

Diffusion sélective : exemple d'utilisation

```
> mpirun -np 4 ./a.out
moi processeur 2 ai recu 1065 1096
moi processeur 3 ai recu 1097 1128
moi processeur 0 ai recu 1001 1032
moi processeur 1 ai recu 1033 1064
```

```
program scatter
  use mpi_f08
  implicit none
  integer, parameter :: nb_valeurs=128
  integer :: nb_procs,rang,nloc,i,ier
  integer,allocatable,dimension(:) :: valeurs,donnees
  call MPI_Init(ier)
  call MPI_Comm_size(MPI_COMM_WORLD,nb_procs,ier)
  call MPI_Comm_rank(MPI_COMM_WORLD,rang,ier)
  nloc=nb_valeurs/nb_procs
  allocate(donnees(nloc))
  if (rang == 2) then
     allocate(valeurs(nb_valeurs))
     valeurs(:)=(/(1000+i,i=1,nb_valeurs)/)
  end if
  call MPI_Scatter(valeurs,nloc,MPI_INTEGER,donnees,nloc,MPI_INTEGER,2,MPI_COMM_WORLD,ier)
  print *,'moi processeur',rang, 'ai recu', donnees(1),donnees(nloc)
```

Les données sont distribuées en tranches égales, dans l'ordre des rangs des processus du communicateur au sein duquel s'effectue la communication collective.

Cas où le nombre d'éléments à diffuser/collecter varie selon le rang du processus courant

- Les sous-programmes MPI_Scatterv(), MPI_Gatherv(), MPI_Allgatherv() et MPI_Alltoallv() étendent MPI_Scatter(), MPI_Gather(), MPI_Allgather() et MPI_Alltoall() au cas où le nombre d'éléments à diffuser ou collecter est différent suivant les processus.

Exemple: Diffusion sélective MPI_Scatterv()

```
integer :: r_count   ! a calculer pour chaque processus
integer, dimension(nbProcs) :: s_counts,s_displs ! uniquement pour sender
call MPI_Scatterv(s_buf, s_counts, s_displs, s_type,
                  r_buf, r_count,           r_type, sce, comm,ier)
```

- r_count : nombre d'éléments à recevoir (varie selon le rang)
- s_counts : tableau de nb d'éléments à envoyer
- s_displs : tableau de déplacements associés

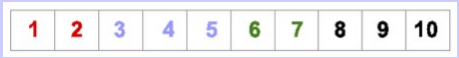
Cas d'un tableau de déplacements indiqué en octets : MPI_Scatterw(), MPI_Gatherw(), MPI_Allgatherw() et MPI_Alltoallw()

Cas où le nombre d'éléments à diffuser = f(rang)

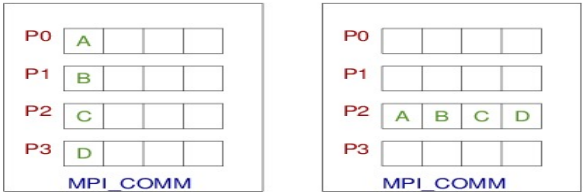
```
> mpirun -np 4 ./a.out
2           3           2           3
0           2           5           7
[ 2 ]: ai recu          6           7
[ 0 ]: ai recu          1           2
[ 3 ]: ai recu          8           9          10
[ 1 ]: ai recu          3           4           5
```

```
program scatterv
  use mpi_f08
  implicit none
  integer, parameter :: nb_valeurs=10
  integer            :: nb_procs,rang,i,ier
  integer            :: r_count
  integer,allocatable,dimension(:) :: s_buf,r_buf
  integer,allocatable,dimension(:) :: s_counts,s_displs
  call MPI_Init(ier)
  call MPI_Comm_size(MPI_COMM_WORLD,nb_procs,ier)
  call MPI_Comm_rank(MPI_COMM_WORLD,rang,ier)
  ! Each MPI processus allocates its own sub-array
  r_count = (nb_valeurs*(rang+1))/nb_procs - (nb_valeurs*rang)/nb_procs
  allocate(r_buf(r_count))
  if (rang == 2) then
    allocate(s_buf(nb_valeurs))
    s_buf(:)=(/(i,i=1,nb_valeurs)/)
    ! Calcul nb d'elements a envoyer et deplacement associe
    allocate(s_counts(nb_procs),s_displs(nb_procs))
    s_counts(1)=nb_valeurs/nb_procs
    s_displs(1)=0
    do i=2,nb_procs
      s_displs(i)=s_displs(i-1)+s_counts(i-1)
      s_counts(i)=(nb_valeurs*i)/nb_procs - (nb_valeurs*(i-1))/nb_procs
    enddo
    print *, s_counts
    print *, s_displs
  endif
  call MPI_Scatterv(s_buf,s_counts,s_displs,MPI_INTEGER,r_buf,r_count,MPI_INTEGER,2,MPI_COMM_WORLD,ier)
  print *,'[',rang, ']: ai recu',r_buf(:)
  call MPI_Finalize(ier)
end program scatterv
```

Exemple de la diffusion d'un tableau de 10 éléments parmi les 4 processus d'un communicateur.



Collecte de données réparties : MPI_Gather




```
call MPI_Gather(send_buffer, send_count, send_type, &
                recv_buffer, recv_count, recv_type, rank, comm, ier)
```

- send_buffer / recv_buffer : adresse de début du buffer d'envoi / de réception
- send_count / recv_count : nombre d'éléments à envoyer à / recevoir de chaque processus
- send_type / recv_type : type MPI de données envoyées / reçues
- rank : rang du processus qui collecte les données

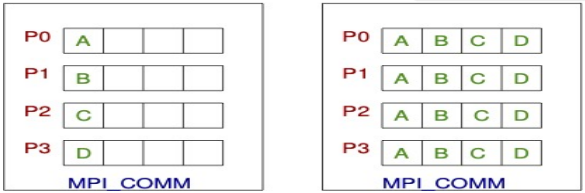
Collecte de données réparties : exemple d'utilisation

```
> mpirun -np 4 ./a.out
 [2] ai recu 1001 1032 1033 1128
```

```
program gather
 implicit none
 use mpi_f08
 integer, parameter :: nb_valeurs=128
 integer :: nb_procs,rang,nloc,i,ier
 integer,allocatable,dimension(:) :: valeurs
 integer,dimension(nb_valeurs) :: donnees
 call MPI_Init(ier)
 call MPI_Comm_size(MPI_COMM_WORLD,nb_procs,ier)
 call MPI_Comm_rank(MPI_COMM_WORLD,rang,ier)
 nloc=nb_valeurs/nb_procs
 allocate(valeurs(nloc))
 valeurs(:)=(/(1000+rang*nloc+i,i=1,nloc)/)
 call MPI_Gather(valeurs,nloc,MPI_INTEGER,donnees,nloc,MPI_INTEGER,2,MPI_COMM_WORLD,
 if (rang == 2) print *,'[2] ai recu ',donnees(1),&
      donnees(nloc),donnees(nloc+1),donnees(nb_valeurs)
 call MPI_Finalize(ier)
```

Collecte générale de données réparties : MPI_Allgather

Correspond à un MPI_Gather suivi d'un MPI_Bcast.




```
call MPI_Allgather(send_buffer, send_count, send_type, &
                   recv_buffer, recv_count, recv_type, comm, ier)
```

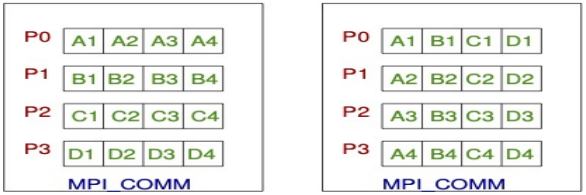
Remarque : Par rapport à MPI_Gather on ne précise plus le rang du processus qui fait la collecte.

Collecte générale de données réparties : exemple d'utilisation

```
> mpirun -np 4 ./a.out
[0] a recu 1001 1032 1033 1128
[1] a recu 1001 1032 1033 1128
[2] a recu 1001 1032 1033 1128
[3] a recu 1001 1032 1033 1128
```

```
program Allgather
 implicit none
 use mpi_f08
 integer, parameter :: nb_valeurs=128
 integer :: nb_procs,rang,nloc,i,ier
 integer,allocatable,dimension(:) :: valeurs
 integer,dimension(nb_valeurs) :: donnees
 call MPI_Init(ier)
 call MPI_Comm_size(MPI_COMM_WORLD,nb_procs,ier)
 call MPI_Comm_rank(MPI_COMM_WORLD,rang,ier)
 nloc=nb_valeurs/nb_procs
 allocate(valeurs(nloc))
 valeurs(:)=(/(1000+rang*nloc+i,i=1,nloc)/)
 call MPI_Allgather(valeurs,nloc,MPI_INTEGER, donnees,nloc,MPI_INTEGER,MPI_COMM_WORLD
 print *,'[',rang,'] a recu ',donnees(1),donnees(nloc),&
      donnees(nloc+1),donnees(nb_valeurs)
 call MPI_Finalize(ier)
```

Echange général de données : MPI_Alltoall




```
call MPI_Alltoall(send_buffer, send_count, send_type, &
                  recv_buffer, recv_count, recv_type, comm, ier)
```

Remarque : les arguments sont les mêmes que pour MPI_Allgather mais on croise les données échangées.
Remarque : il est possible d'utiliser le même buffer pour l'envoi et la réception, dans ce cas le mot clé MPI_IN_PLACE doit être utilisé pour l'argument send_buffer.

Échange général : exemple d'utilisation

```
> mpirun -np 3 ./a.out |sort
Avant   0/3 avec tab=    1   2   3
Avant   1/3 avec tab=   11  12  13
Avant   2/3 avec tab=   21  22  23
Ensuite 0/3 avec tab=    1  11  21
Ensuite 1/3 avec tab=    2  12  22
Ensuite 2/3 avec tab=    3  13  23
```

```
PROGRAM all2all
use mpi_f08
implicit none
integer:: rank, taille, ierr, i
integer, allocatable:: tab(:)
character(len=64)::form
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
call MPI_Comm_size(MPI_COMM_WORLD,taille,ierr)
allocate(tab(taille))
tab(:)=(/(10*rank+i,i=1,taille)/)
write(form,'(a,i0,a)') "(2(a,i0),a,",taille,"(x,i3))"
write(6,form),"Avant   ",rank,"/",taille," avec tab= ",tab(:)
call MPI_Alltoall(MPI_IN_PLACE, 1, MPI_INTEGER, tab, 1, MPI_INTEGER,MPI_COMM_WORLD,ierr)
write(6,form),"Ensuite ",rank,"/",taille," avec tab= ",tab(:)
call MPI_Finalize(ierr)
END PROGRAM all2all
```

Pour une programmation correcte si les buffers d'envoi et réception sont identiques:

call MPI_Alltoall(MPI_IN_PLACE , 1, MPI_INTEGER, tab, 1, MPI_INTEGER, MPI_COMM_WORLD, ierr)

Réductions réparties

- Réduction des données réparties sur un ensemble de processus avec récupération du résultat.
  - Résultat sur un seul processus: MPI_Reduce().
  - Résultat sur tous les processus: MPI_Allreduce().
- Chaque processus doit travailler sur la même taille de buffer
- L'opérateur s'applique élément à élément du buffer
- Réduction:
  - opération arithmétique ou logique prédéfinie par MPI (somme,max,min des éléments d'un vecteur),
  - définie par le programmeur : MPI_OP_CREATE et MPI_OP_FREE

Opérations classiques de réduction

Nom	Opération
MPI_SUM	Somme des éléments
MPI_PROD	Produit
MPI_MAX	Recherche du maximum
MPI_MIN	Recherche du minimum
MPI_MAXLOC	Recherche de l'indice du maximum
MPI_MINLOC	// minimum
MPI_LAND	ET logique
MPI_LOR	OU logique
MPI_LXOR	OU exclusif logique

Réductions réparties : syntaxe

```
call MPI_Reduce(myvar, var, nb, MPI_REAL, MPI_SUM, rang_reduc,&
                MPI_COMM_WORLD, ier)
```

- myvar: buffer local
- var : buffer réduit
- nb : nombre d'éléments dans chacun des buffers
- *MPI_SUM* : type d'opération
- rang_reduc : rang du processus qui fait la réduction

Réductions réparties : exemple d'utilisation

```
> mpirun -np 3 ./reduce
Processus 1/3 avec tab  =   11  12  13  14
Processus 1/3 avec reduc=    0   0   0   0
Processus 2/3 avec tab  =   21  22  23  24
Processus 2/3 avec reduc=   33  36  39  42
Processus 0/3 avec tab  =    1   2   3   4
Processus 0/3 avec reduc=    0   0   0   0
```

```
1  PROGRAM reduc
2  use mpi_f08
3  implicit none
4    integer:: rank, taille, ierr, i
5    integer:: tab(4), reduc(4)=0
6    character(len=64)::form
7    call MPI_Init(ierr)
8    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
9    call MPI_Comm_size(MPI_COMM_WORLD, taille, ierr)
0    tab(:)=(/(10*rank+i,i=1,4)/)
1    write(form,'(a,i0,a)') "(2(a,i0),a,",4,"(x,i3))"
2    write(6,form),"Processus ",rank,"/",taille," avec tab= ",tab(:)
3    call MPI_Reduce(tab, reduc, 4, MPI_INTEGER, MPI_SUM, 2, MPI_COMM_WORLD, ierr)
4    write(6,form),"Processus ",rank,"/",taille," avec reduc= ",reduc(:)
5    call MPI_Finalize(ierr)
6  END PROGRAM reduc
```

Réductions générale en place : exemple d'utilisation

```
> mpirun -np 3 ./reduce
Processus 1/3 avec tab local=    11  12  13  14
Processus 1/3 avec tab réduit=   33  36  39  42
Processus 2/3 avec tab local=    21  22  23  24
Processus 2/3 avec tab réduit=   33  36  39  42
Processus 0/3 avec tab local=     1   2   3   4
Processus 0/3 avec tab réduit=   33  36  39  42
```

```
1  PROGRAM reduceAllinPlace
2  use mpi_f08
3  implicit none
4    integer:: rank, taille, ierr, i
5    integer:: tab(4)
6    character(len=64)::form
7    call MPI_Init(ierr)
8    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
9    call MPI_Comm_size(MPI_COMM_WORLD, taille, ierr)
0    tab(:)=(/(10*rank+i,i=1,4)/)
1    write(form,'(a,i0,a)') "(2(a,i0),a,",4,"(x,i3))"
2    write(6,form),"Processus ",rank,"/",taille," avec tab local=  ",tab(:)
3    call MPI_Allreduce(MPI_IN_PLACE, tab, 4, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD, ierr)
4    write(6,form),"Processus ",rank,"/",taille," avec tab réduit= ",tab(:)
5    call MPI_Finalize(ierr)
6
```

- Tous les processus exécutent la réduction (pas de root)
- Le buffer local est remplacé par MPI_IN_PLACE
- le buffer réduit est le buffer local

Quelques conseils pour conclure

- Ne cherchez pas à optimiser lors de la première écriture $\Rightarrow$ utilisez les appels bloquants, dans un premier temps.
- Utilisez de préférence les communications collectives.

Trainings - MPI - Communications entre processus

Rendez-vous dans le projet habituel pour les exercices sur les communications entre processus.

1 Introduction

2 Premiers pas avec MPI

3 Communications

4 Types dérivés

- Motivations
- Définition des types dérivés
- Création d'un type dérivé
- Conclusions

5 Groupes, communicateurs et topologies

6 Autres fonctionnalités

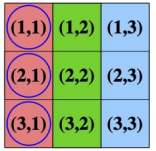
Rappel sur les types de base MPI

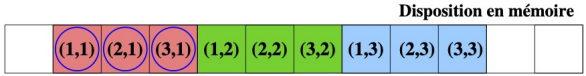
- MPI founit un certain nombre de types de base **prédéfinis** : MPI_INTEGER, MPI_REAL, ...
- Un message peut contenir plusieurs données de ces types de base si :
  - elles sont de même type (homogènes),
  - elles sont contiguës en mémoire.

Exemple : envoi de la première colonne d'un tableau fortran

```
real :: a(3,3)
! ...
call MPI_Send(a, 3, MPI_REAL,...)
```

En fortran les données d'un tableau 2D sont rangées par colonne $\neq$ du C/C++ où elles le sont par ligne.





Rappel sur les types de base MPI

- Problème: comment envoyer une ligne d'un tableau 2D Fortran ?
  - Les données sont de même type (homogènes),
  - Les données ne sont plus contiguës en mémoire .

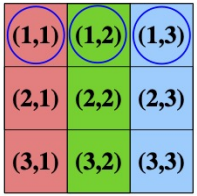


Tableau Fortran

Disposition en mémoire

Stratégies pour contourner ces difficultés

- Effectuer plusieurs envois successifs.
  Peut devenir très coûteux (latence à chaque appel).
- Créer, remplir et envoyer un buffer (MPI_Pack). Coût mémoire (buffer) + Coût CPU (copier/extraire les données dans/depuis le buffer)
- La bonne solution : utiliser des types dérivés MPI .

Les types dérivés permettent de :

- Transmettre des données non contiguës en mémoire,
- Transmettre des structures de données plus complexes ,
- Construire de nouveaux types MPI correspondants aux types dérivés utilisateurs (structures, objets).
- Améliorer la lisibilité du code

Une fois construits, ils peuvent être utilisés pour plusieurs envois.

Exemples de types dérivés

- Un bloc de données homogènes contiguës en mémoire, une colonne de matrice

```
real(kind=8) :: a(10,10)
! ...
call MPI_Send(a, 10, MPI_DOUBLE_PRECISION, ...)
! appel equivalent  : definition d'un type 'mpi_colonne' puis envoi :
call MPI_Send(a, 1, mpi_colonne,...)
```

- Une structure complexe

```
type :: maStruct
  character(20) :: nom
  real          :: contenu
end type maStruct
maStruct :: a ! Instanciation et initialisation d'un objet maStruct
! Definition d'un type derive 'mpi_maStruct' pour le type maStruct
! ...
! Envoi de a
call MPI_Send(a, 1, mpi_maStruct, ...)
```

Déclaration d'un type dérivé

En fortran

```
TYPE(MPI_Datatype) :: mpi_maStruct
```

En C/C++

```
MPI_Datatype mpi_maStruct;
```

Définition et utilisation d'un type dérivé

Un type dérivé est constitué de plusieurs composants : 3 tableaux contenant

- La **longueur** de chaque composant.
- La localisation **(adresse mémoire)** de chaque composant (procédure MPI_Get_address).
- Le **type** de chaque composant.

Mise en œuvre :

1. Déclarer et initialiser les composants.
2. **Définir** le nouveau type MPI, appel à une fonction spécifique : MPI_Type_xxx
3. **Valider** ce type pour les procédures de communication : MPI_Type_commit
4. **Libérer** un type MPI : MPI_Type_free

Création d'un type dérivé

Différentes fonctions sont disponibles. Le choix est déterminé par l'*organisation en mémoire* des données que l'on souhaite échanger.

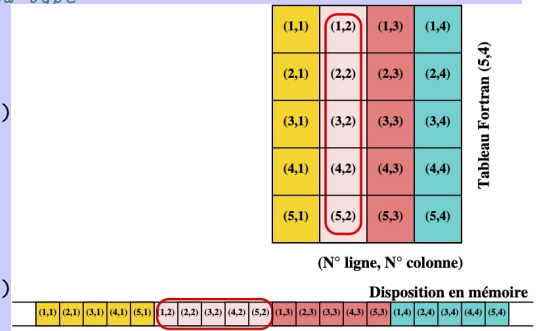
- Données homogènes contiguës en mémoire : MPI_Type_contiguous
- Données homogènes non contiguës en mémoire espacées à pas constant : MPI_Type_vector (MPI_Type_create_hvector si le pas est donné en octet (byte))
- Blocs de données homogènes de longueur variable et espacés à pas variable : MPI_Type_indexed (MPI_Type_create_hindexed)
- Blocs de données non homogènes, de longueur et de pas variables : MPI_Type_create_struct

Données homogènes contiguës en mémoire

```
MPI_Type_contiguous(count, oldtype, newtype, ierror)
```

Un exemple simple pour commencer : type colonne de matrice en fortran

```
type(MPI_Datatype) :: colonne ! le "nom" du nouveau type
real, dimension(5,4) :: a, b
a(:,:) = ...
! Definition du type mpi colonne (5 real contigus)
call MPI_Type_contiguous(5, MPI_REAL, colonne, err)
! Validation du type
call MPI_Type_commit(colonne,err)
! Envoi de la seconde colonne de a(:,:)
if (rg==src) call MPI_Send(a(1,2), 1, colonne,...)
! Reception en colonne 1 de b(:,:)
if (rg==dest) call MPI_Recv(b(1,1), 1, colonne,...)
! Liberation du type
call MPI_Type_free(colonne, err)
```



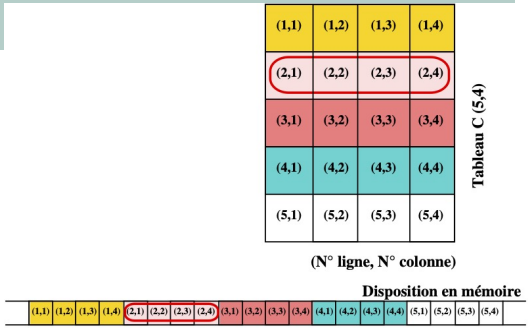
Attention: le type doit être créé par tous les processus utilisateurs

Données homogènes contiguës en mémoire

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Le même exemple : type ligne de matrice en C/C++

```
MPI_Datatype ligne;
ierr = MPI_Type_contiguous(4, MPI_FLOAT, &ligne);
```

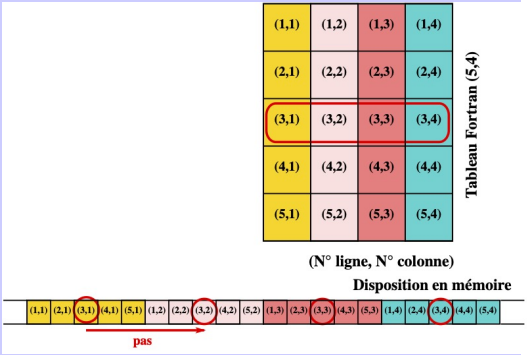


Données homogènes non contiguës en mémoire

```
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
```

Exemple : type ligne de matrice en Fortran

```
TYPE(MPI_Datatype) :: ligne
real, dimension(5,4):: a
...
call MPI_Type_vector(4,1,5,MPI_REAL,ligne,ierr)
call MPI_Type_commit(ligne, ierr)
....
if (rank == 0) then ! envoi de la ligne 3
  call MPI_Send(a(3,1), 1, ligne, 1, &
                tag, MPI_COMM_WORLD, ierr)
else                ! reception en ligne 1
  call MPI_Recv(a(1,1), 1,  ligne, 0, &
                tag, MPI_COMM_WORLD, &
                MPI_STATUS_IGNORE, ierr)
end if
call MPI_Type_free(ligne,ierr)
```

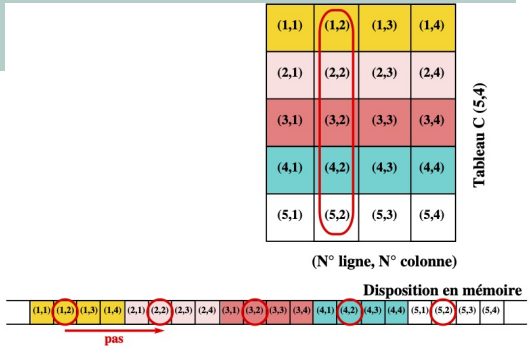


Données homogènes non contiguës en mémoire

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Le même exemple : type colonne de matrice en C/C++

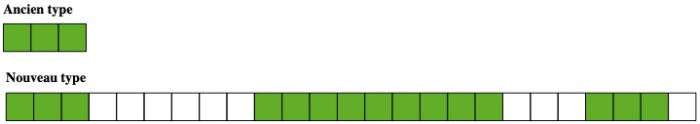
```
MPI_Datatype colonne;
ierr = MPI_Type_vector(5, 1, 4,
       MPI_FLOAT, &colonne);
```



Données homogènes, non contiguës en mémoire, pas irrégulier

```
int MPI_Type_indexed(int count,int blocklengths[], int displacements[],
                     MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Exemple :



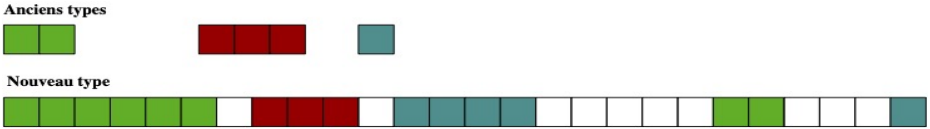

```
int count=3;                          // nombre de blocs au total
int blocklengths[count]=(1,3,1);      // Taille de chacun des blocs (en nbr d'elements)
int displacements[count]=(0,3,7);     // Offset des blocs par rapport au debut du buffer
MPI_Type_indexed(count,blocklengths,displacements,oldtype, &newtype);
```

- Les tailles des blocs sont calculées en nombre d'éléments du type initial.
- Les déplacements sont calculés en nombre d'éléments du type initial (indexed) ou en octets (create_hindexed) et dans ce cas sont de type MPI_Aint (C/C++) ou INTEGER(KIND=MPI_ADDRESS_KIND) (Fortran).

Cas général : données hétérogènes

```
MPI_Type_create_struct(int count,int blocklengths[],MPI_Aint displacements[],
                       MPI_Datatype oldtypes[],MPI_Datatype *newtype)
```

Exemple :




```
count=5, blocklengths=(3,1,4,1,1), displacements=(0,7,11,20,25)
oldtypes(type1,type2,type3,type1,type3)
MPI_Type_create_struct(count,blocklengths,displacements, oldtypes, &newtype)
```

- Déplacements calculés à partir des différences d'adresse relativement à l'adresse de départ
- Le tableau des déplacements est de type MPI_Aint (C) ou integer(KIND=MPI_ADDRESS_KIND) (Fortran)

Résumé - Compléments (1/2)

- Valider la création d'un nouveau type

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

- Libérer le type crée lorsqu'il n'est plus utilisé

```
int MPI_Type_free(MPI_Datatype *datatype)
```

- Taille totale d'un type de données (en octets)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

Résumé - Compléments (2/2)

- Pour connaître l'adresse (exprimée en octet) d'un élément (d'une localisation en mémoire), utile pour le calcul des déplacements ...

```
int MPI_Get_address(const void *location, MPI_Aint *address)
```

Attention aux types pour les déclarations d'adresse mémoire:

```
MPI_Aint address;  // En C/C++
```

```
integer(kind=MPI_ADDRESS_KIND) :: address  ! En fortran
```

- Calcul des déplacements en octet relativement à l'adresse de départ

```
for (unsigned int i=0; i<nb ; i++)
  deplacements[i] = address[i]-address[0];
```

- Etendue et borne inférieure d'un type MPI

```
MPI_Aint borne_inf, etendue;
ierr = MPI_Type_get_extent(MPI_datatype, &borne_inf, &etendue)
```

Conclusions

- Outils puissants et portables de description de données.
- Correspondances des types dérivés MPI avec les types abstraits créés par l'utilisateur.
- Ils rendent les échanges de données :
  - plus simples et plus lisibles,
  - plus performants (moins de recopies des données),
- L'association des topologies et des types dérivés simplifie l'écriture des problèmes de décomposition de domaine

Trainings - MPI - Types dérivés

Rendez-vous dans le projet habituel pour les exercices sur les types dérivés.

1 Introduction

2 Premiers pas avec MPI

3 Communications

4 Types dérivés

5 Groupes, communicateurs et topologies
- Rappels sur les communicateurs
- Construction et manipulations de groupes de processus, de communicateurs
- Topologies
- Intra et inter-communicateurs

6 Autres fonctionnalités

Rappels : définition et propriétés d'un communicateur

Communicateur : un ensemble de processus susceptibles de communiquer entre eux.

⇒ Un contexte de communication.

⇒ Un groupe de processus ordonnés.

- Un communicateur par défaut créé à l'initialisation, *MPI_COMM_WORLD*, qui contient tous les processus actifs.
- A tout instant on peut connaître :
  - le nombre de processus gérés par un communicateur, *MPI_Comm_size*
  - le rang de chaque processus dans ce communicateur, *MPI_Comm_rank*

Rappels : définition et propriétés d'un communicateur

- Les procédures de communication se font à l'intérieur d'un commmunicateur, qui intervient toujours comme argument de ces routines. Par exemple,
  point à point :

```
MPI_Send(data, send_count, send_type, dest, tag, MPI_COMM, err)
```

  collectives :

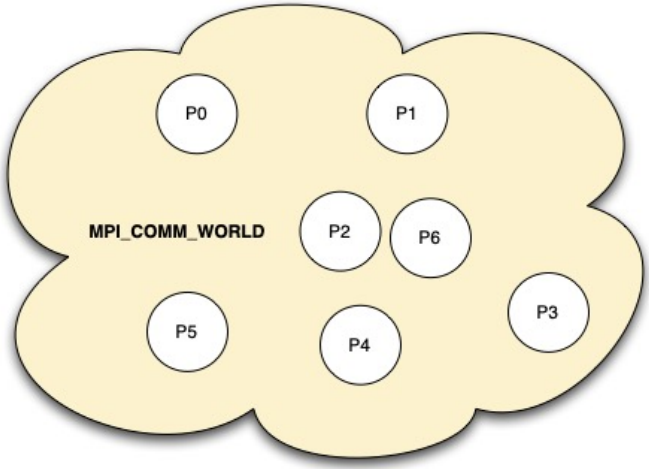
```
MPI_Bcast(data, send_count, send_type, rank, MPI_COMM, err)
```

Pourquoi définir de nouveaux communicateurs ?

Pour échanger des messages, des processus doivent nécessairement appartenir au même communicateur. Le communicateur définit le contexte de communication. Il donne un " cadre " aux échanges et fixe la portée des communications entre les processus du groupe.
Créer de nouveaux communicateurs permettra :

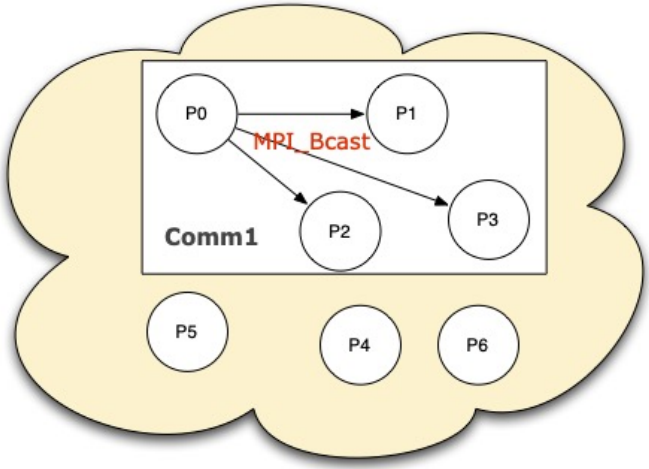
- de spécialiser un sous-groupe de processus. Par exemple pour appeler une routine de communication collective dans ce sous-groupe,
- d'organiser un certains nombre de processus. Par exemple pour mieux "coller" au problème à traiter en réordonnant les processus.

Exemples



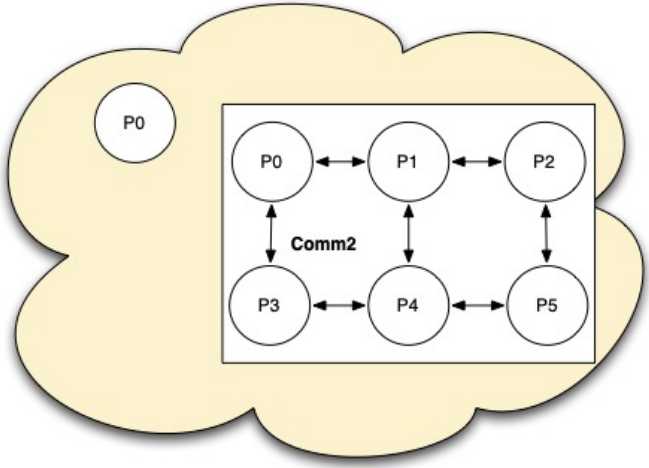
Exemples

Réduction de la portée d'une communication collective , spécialisation d'un sous groupe :



Exemples

Ré-organisation des processus : topologie, notion de voisins ...



Communicateurs

Deux méthodes pour construire un communicateur :

- à partir d'un communicateur existant (découpage, ...),
- à partir d'un groupe de processus.

Deux grands types de communicateurs :

- intra-communicateur : définit un contexte de communication entre processus (inclus dans *MPI_COMM_WORLD*), implique un seul groupe de processus.
- inter-communicateur : définit un contexte de communication entre différents communicateurs, implique deux groupes de processus.

Construction et manipulations de groupes de processus (1)

Groupe : ensemble de processus ordonnés. Communicateur ⇒ groupe.

Type : *Type(MPI_Group)* en Fortran, *MPI_Group* en C.

- Obtenir le groupe de processus d'un communicateur :

```
MPI_Comm_group(comm_source, group)
```

- Construction à partir d'un autre groupe :

```
// target créée par inclusion des processus 'ranks' du groupe 'source'
MPI_Group_incl(source, dim, ranks, target)
// target crée par exclusion des processus 'ranks' du groupe 'source'
MPI_Group_excl(source, dim, ranks, target)
```

dim : nombre de processus dans le groupe (dans le groupe target) *ranks* : liste des rangs (dans source) des processus à inclure/exclure dans/de target.

Construction et manipulations de groupes de processus (2)

- Construction à partir de plusieurs groupes (opérations ensemblistes) :

```
MPI_Group_union(groupe1, groupe2, new_groupe)
MPI_Group_intersection(groupe1, groupe2, new_groupe)
MPI_Group_difference(groupe1, groupe2, new_groupe)
```

- Destruction:

```
MPI_Group_free(group)
```

- Informations :

```
MPI_Group_rank(groupe, rang_dans_le_groupe) // Rang du proc. courant dans le groupe
MPI_Group_size(groupe, taille_du_groupe)    // Taille du groupe
// Calcul des rangs dans group2 à partir de ceux dans group1
MPI_Group_translate_ranks(group1, dim, ranks_in_1, group2, ranks_in_2)
MPI_Group_compare(group1, group2, result) // Comparaison de deux groupes
```

result : MPI_IDENT (mêmes procs/ordre), MPI_SIMILAR (mêmes procs), MPI_UNEQUAL

Comment construire un nouveau communicateur?

Il existe deux méthodes pour créer un nouveau communicateur.

1. Créer un groupe de processus puis un communicateur à partir de ce groupe,

```
MPI_Comm_create(comm_source, group_source, new_comm)

// Group_source : sous-groupe de processus de comm_source
```

   Voir exemple en TP.

2. Partir d'un communicateur existant et le découper et/ou le réorganiser. Voir ci-après, *MPI_Comm_split* etc

Intérêt de cette dernière méthode?

- Eviter de passer par des groupes.
- Eviter les tests conditionnels dans les routines (voir TP).

Contruction d'un nouveau communicateur à partir d'un communicateur existant

Instructions de création

- copie d'un communicateur : *MPI_Comm_dup*
- découpage d'un communicateur : *MPI_Comm_split*
- création d'une grille de processus (topologie) : *MPI_Cart_create*
- découpage d'une topologie : *MPI_Cart_sub*

Instruction de libération : *MPI_Comm_free*

Duplication d'un communicateur

```
MPI_Comm_dup(Comm_origine,nouveau_comm)
```

Le nouveau communicateur possède le même groupe de processus mais avec un nouveau contexte de communication.

Quel intérêt?

- Créer un communicateur privé (au début d'une nouvelle librairie parallèle par exemple).
- Séparer les contextes et les messages, "sécuriser" les échanges.

Partitionnement d'un communicateur

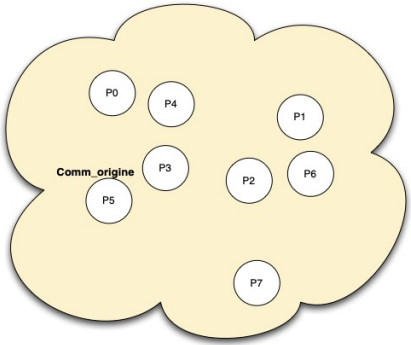
```
MPI_Comm_split(comm_origine, couleurs, clefs, comm)
```

couleurs : définit les sous-ensemble.
clefs : définit l'ordre dans chaque sous-ensemble.

Partitionnement d'un communicateur

```
MPI_Comm_split(comm_origine, couleurs, clefs, comm)
```

couleurs : définit les sous-ensemble.
clefs : définit l'ordre dans chaque sous-ensemble.

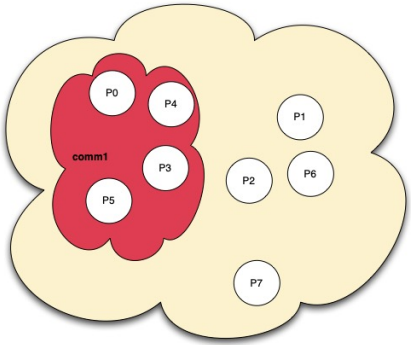


rangs	0	1	2	3	4	5	6	7

Partitionnement d'un communicateur

```
MPI_Comm_split(comm_origine, couleurs, clefs, comm)
```

couleurs : définit les sous-ensemble.
clefs : définit l'ordre dans chaque sous-ensemble.

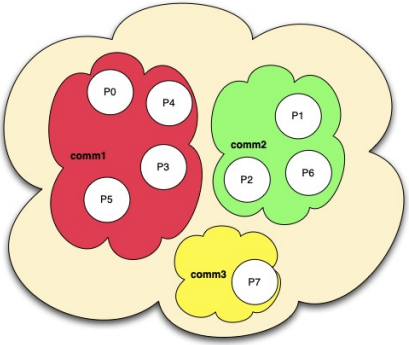


rangs	0	1	2	3	4	5	6	7
couleurs	1			1	1	1		

Partitionnement d'un communicateur

```
MPI_Comm_split(comm_origine, couleurs, clefs, comm)
```

couleurs : définit les sous-ensemble.
clefs : définit l'ordre dans chaque sous-ensemble.

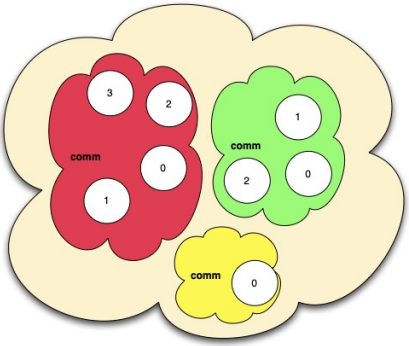


rangs	0	1	2	3	4	5	6	7
couleurs	1	4	4	1	1	1	4	7

Partitionnement d'un communicateur

```
MPI_Comm_split(comm_origine, couleurs, clefs, comm)
```

couleurs : définit les sous-ensemble.
clefs : définit l'ordre dans chaque sous-ensemble.

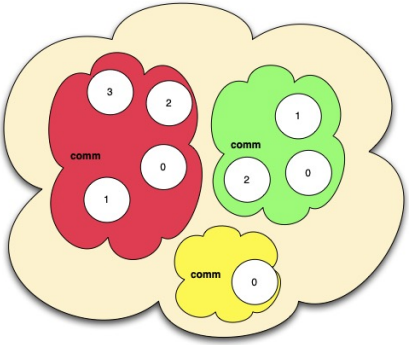


rangs	0	1	2	3	4	5	6	7
couleurs	1	4	4	1	1	1	4	7
clés	3			0	2	1		

Partitionnement d'un communicateur

```
MPI_Comm_split(comm_origine, couleurs, clefs, comm)
```

couleurs : définit les sous-ensemble.
clefs : définit l'ordre dans chaque sous-ensemble.



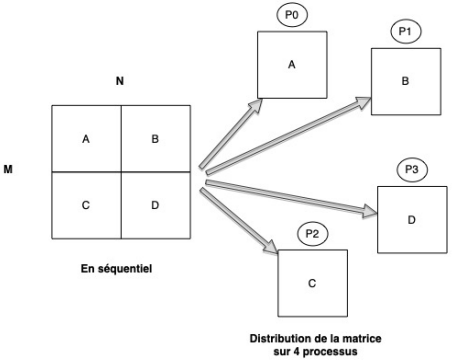
rangs	0	1	2	3	4	5	6	7
couleurs	1	4	4	1	1	1	4	7
clés	3	1	2	0	2	1	0	7

Topologies cartésiennes

Réorganisation des processus d'un communicateur selon une grille cartésienne .

Contexte et motivations :

- Méthodes de décomposition de domaine.
- La correspondance entre les grilles de données et les grilles de processus favorise les performances et facilite l'implémentation.



Création d'une topologie cartésienne

MPI_Cart_create : définition d'un nouveau communicateur de type "grille de processus", caractérisé par :

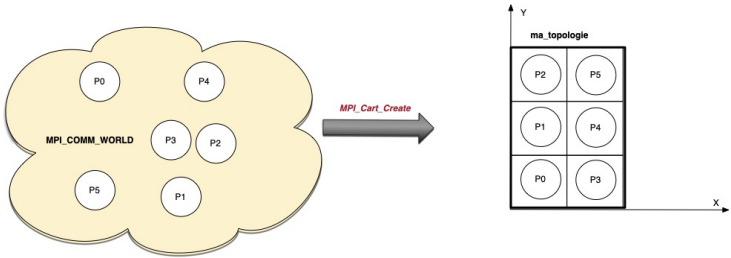
- sa dimension ,
- sa résolution , le nombre de processus dans chaque direction,
- sa périodicité .

Création d'une topologie cartésienne

MPI_Cart_create : définition d'un nouveau communicateur de type "grille de processus", caractérisé par :

- sa dimension ,
- sa résolution , le nombre de processus dans chaque direction,
- sa périodicité .

```
// dimension = 2, resolution = (2,3) period = (True, False), reorganise = True
MPI_Cart_create(MPI_COMM_WORLD, dimension, resolution, period, reorganise, ma_topologie)
```

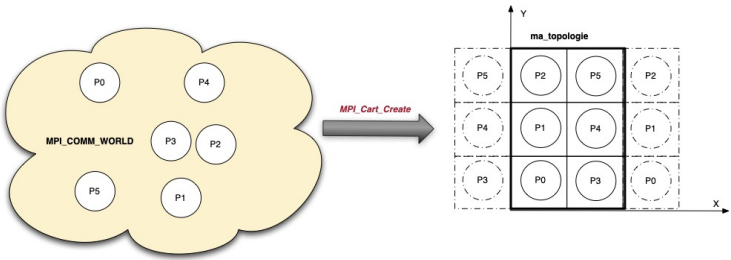


Création d'une topologie cartésienne

MPI_Cart_create : définition d'un nouveau communicateur de type "grille de processus", caractérisé par :

- sa dimension ,
- sa résolution , le nombre de processus dans chaque direction,
- sa périodicité .

```
// dimension = 2, resolution = (2,3) period = (True, False), reorganise = True
MPI_Cart_create(MPI_COMM_WORLD, dimension, resolution, period, reorganise, ma_topologie)
```



Création d'une topologie cartésienne

MPI_Cart_create : définition d'un nouveau communicateur de type "grille de processus", caractérisé par :

- sa dimension ,
- sa résolution , le nombre de processus dans chaque direction,
- sa périodicité .

```
// dimension = 2, resolution = (2,3) period = (True, False), reorganise = True
MPI_Cart_create(MPI_COMM_WORLD, dimension, resolution, period, reorganise, ma_topologie)
```

- ⚠ routine collective et bloquante !
- Si le paramètre *reorganise* est faux, le rang des processus dans *comm* est conservé dans la nouvelle topologie. Dans le cas contraire, MPI peut réorganiser les processus, par exemple pour mieux coller à l'architecture physique de la machine.

Comment fixer la résolution?

- Directement

```
integer, dimension (2) :: resolution
resolution(1) = 4
resolution(2) = 2
```

⚠ la résolution doit être “compatible” avec la taille du communicateur.

- En utilisant la fonction MPI_Dims_create qui laisse l'interface MPI choisir une répartition optimale, en fonction du nombre de processus disponibles.

```
call MPI_Dims_create(nombre_de_processus, dimensionTopo, resolution, erreur)
```

dimensionTopo est un argument d'entrée et de sortie qui permet de contrôler partiellement la distribution. Toute valeur différente de zéro sera gardée (si possible), un zéro laisse MPI choisir.

resolution (entrée)	appel MPI_Dims_create	resolution (sortie)
(0,0)	(8,2,dims)	(4,2)
(0,0,0)	(16,3,dims)	(4,2,2)
(0,4,0)	(16,3,dims)	(2,4,2)
(0,3,0)	(16,3,dims)	erreur

Propriétés des processus dans la topologie

Rang dans la topologie : *MPI_Comm_rank* ou *MPI_Cart_rank* (donne le rang du processus à partir de ses coordonnées).

```
call MPI_Cart_rank(topo_name, coords, rang)
```

Coordonnées dans la topologie : *MPI_Cart_coords*. Donne les coordonnées du processus dans la grille à partir de son rang.

```
MPI_Cart_coords(nomTopo, rang, dimTopo, coords)
```

- Coords est un vecteur, sa taille est la dimension de la topologie.
- Les indices des coordonnées démarrent à 0, quel que soit le langage.

Propriétés des processus dans la topologie

Voisins .

Une routine permet de récupérer le rang des voisins du processus courant, pour chaque direction : *MPI_Cart_shift*

```
MPI_Cart_shift(topo_name, direction, step, previous_rank, next_rank)
```

- *direction* indique l'axe de déplacement (0:x, 1:y ...), *step* donne la taille du déplacement.
- Les rangs des voisins sont récupérés dans *previous_rank* et *next_rank*.
- La routine prend en compte la périodicité éventuelle de la topologie.
- Si le processus n'a pas de voisin dans une direction, la routine renvoie *MPI_PROC_NULL* ou *MPI_UNDEFINED* .

Découpage d'une topologie cartésienne

Il peut parfois être utile de dégénérer une topologie cartésienne (i.e. de créer des sous topologies de dimension inférieure à la topologie d'origine).

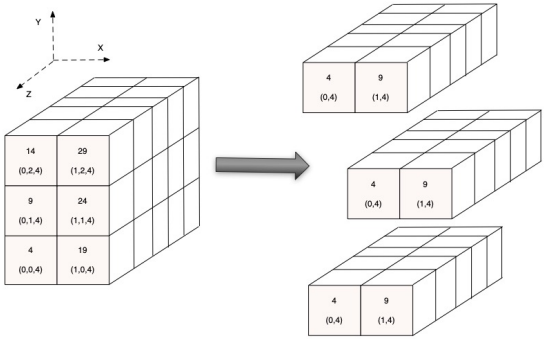
Comment?

- *MPI_Comm_split* (voir plus haut).
- *MPI_Cart_sub* : le découpage va se faire selon une ou plusieurs directions (au max N-1, N étant la dimension de la topologie).

Découpage d'une topologie cartésienne

Exemple : on découpe une topologie 3D en plans :

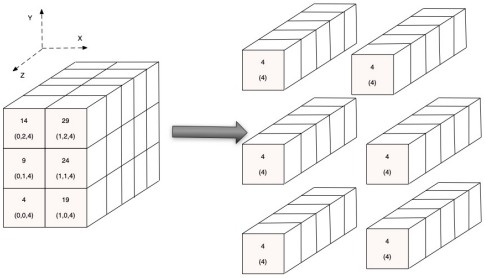
```
keep_directions = (/true,false,true/)  ! Do not split dir 0 and 2. Split 1.
call MPI_Cart_sub(source_topology_3d, keep_directions, new_topology_2d, error)
```



Découpage d'une topologie cartésienne

Exemple : la même topologie, découpée en crayons (topologies 1D) :

```
keep_directions = (/false,false,true/) ! Do not split dir 2. Split 0 and 1.
call MPI_Cart_sub(source_topology_3d, keep_directions, new_topology_1d, error)
```



Attention : une seule variable (nomTopo1D) qui représente 6 communicateurs différents, selon le rang du processus courant.

Inter- communicateurs

- Les processus de deux intra-communicateurs distincts ne peuvent communiquer entre eux que s'il existe un lien de communication entre ces deux intra-communicateurs.
- Un inter-communicateur est un communicateur qui établit un pont entre deux intra-communicateurs.

Création d'un inter-communicateur

On établit un pont entre un groupe local et un groupe distant ("local" et "distant" étant relatifs au groupe auquel appartient le processus courant).

```
MPI_Intercomm_create(local_comm, local_leader, bridge, remote_leader, tag, new_comm)
```

local_leader et *remote_leader* sont les rangs (respectivement dans local_comm et dans bridge) des processus qui établissent le pont entre les groupes.
bridge DOIT contenir local_leader et remote_leader.

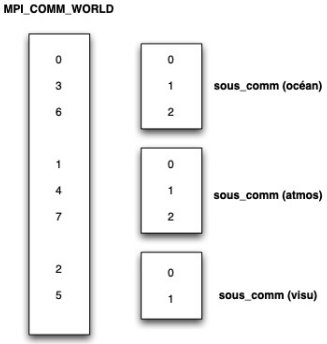
Exemple d'utilisation d'inter-communicateurs

Pour illustrer la création et l'utilisation d'inter-communicateurs, on présente un cas classique (voir MPI forum, cours idris ...) de couplage de code "océan-atmosphère". On considère donc qu'on a deux librairies parallèles, l'une utilisant un modèle atmosphérique et l'autre un modèle océanique, complétées par un module de visualisation.

Exemple d'utilisation d'inter-communicateurs

On commence par répartir les processus entre les trois modèles :

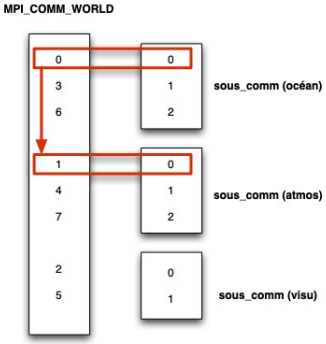
```
call MPI_Comm_rank(MPI_COMM_WORLD,rangMonde,erreur)
call MPI_Comm_size(MPI_COMM_WORLD,nbProcs,erreur)
ocean = 0
atmosphere = 1
visu = 2
couleur = mod(rang,3)
call MPI_Comm_split(MPI_COMM_WORLD,couleur,rangMonde,&
sous_comm,erreur)
```



Exemple d'utilisation d'inter-communicateurs

Puis on construit les inter-communicateurs.

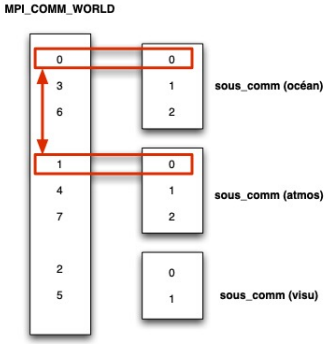
```
select case(couleur)
 case(ocean)
 call MPI_Intercomm_create(sous_comm,0, &
 MPI_COMM_WORLD,1, tag1, oceanAtmo,erreur)
```



Exemple d'utilisation d'inter-communicateurs

Puis on construit les inter-communicateurs.

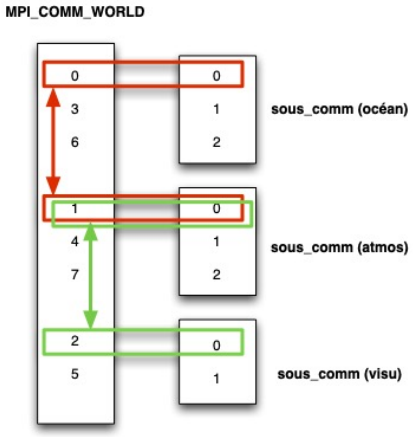
```
select case(couleur)
 case(ocean)
 call MPI_Intercomm_create(sous_comm,0, &
 MPI_COMM_WORLD,1, tag1, oceanAtmo,erreur)

 case(atmosphere)
 call MPI_Intercomm_create(sous_comm,0, &
 MPI_COMM_WORLD,0, tag1, oceanAtmo,erreur)
```



Exemple d'utilisation d'inter-communicateurs

Puis on construit les inter-communicateurs.

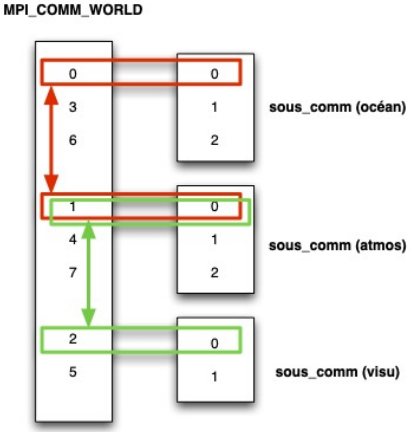
```
select case(couleur)
 case(ocean)
 call MPI_Intercomm_create(sous_comm,0,&
                  MPI_COMM_WORLD,1,&
                  tag1, oceanAtmo,erreur)

 case(atmosphere)
 call MPI_Intercomm_create(sous_comm,0,&
                  MPI_COMM_WORLD,0,&
                  tag1, oceanAtmo,erreur)
 call MPI_Intercomm_create(sous_comm,0,&
                  MPI_COMM_WORLD,2,&
                  tag2, visuAtmo,erreur)
 case(visu)
 call MPI_Intercomm_create(sous_comm,0,&
                  MPI_COMM_WORLD,1,&
                  tag2, visuAtmo,erreur)
```



Exemple d'utilisation d'inter-communicateurs

Chacun peut ensuite appeler un code particulier

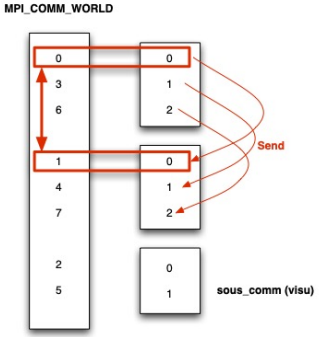
```
select case(couleur)
case(ocean)

call code_ocean(sous_comm,oceanAtmo)

case(atmosphere)

call code_atmo(sous_comm, oceanAtmo,visuAtmo)

case(visu)

call code_visu(sous_comm,oceanAtmo,visuAtmo)
```



Exemple d'utilisation d'inter-communicateurs

envoi d'un message de atmo vers visu

```
subroutine code_ocean(sous_comm, oceanAtmo)
 ! ...
 call MPI_Comm_rank(sous_comm,rang,erreur)
 call MPI_Send(a,size_a, MPI_REAL,rang,tag,&
               oceanAtmo,erreur)

end subroutine
subroutine code_atmo(sous_comm, oceanAtmo,visuAtmo)
 ! ...
 call MPI_Comm_rank(sous_comm,rang,erreur)
 call MPI_Recv(a,size_a,MPI_REAL,rang,tag,&
               oceanAtmo,statut,erreur)

end subroutine
```



Un autre exemple d'utilisation d'inter-communicateurs

Transport de scalaire passif dans un écoulement turbulent. Thèse de J.M. Etancelin, Nov. 2014.

Navier-Stokes/Poisson	**Transport**
$EDP(\omega, v, t, x, y, z) = 0$	$EDP(\rho, v, t, x, y, z) = 0$
Grille cartésienne 3D	Grille cartésienne 3D
$N_x \times N_y \times N_z$	$M_x \times M_y \times M_z$
Découpage MPI 1 ou 2D (crayons/plans)	Résolution multi-GPU, pilotée par MPI.

- problème multi-échelle
- architecture hybride (GPU/CPU)
- v nécessaire aux deux problèmes

Un autre exemple d'utilisation d'inter-communicateurs

On associe un communicateur à chaque tâche :

- n_{cpu} processus dans comm 'CPU' pour résoudre Navier-Stokes/Poisson.
  v^{cpu} : n_{cpu} tableaux locaux de taille $N_x \times N_y \times \frac{N_z}{n_{cpu}}$
- n_{gpu} processus dans comm 'GPU' pour résoudre le problème de transport.
  v^{gpu} : tableaux distribués dont la taille dépend du nombre de GPU.

Chaque tâche fonctionne de manière indépendante (avec éventuellement des communications MPI internes), mais, à chaque itération en temps il faudra synchroniser les valeurs du champ de vitesse

- échanges entre les deux communicateurs,
- redistribution des données,

$\Rightarrow$ utilisation d'un inter-communicateur pour connecter comm CPU et comm GPU.

Compléments

Pour savoir si comm est un inter (flag = vrai) ou un intra-communicateur :

```
call MPI_Comm_test_inter(comm,flag,error)
```

Pour connaitre le nombre de processus dans le communicateur distant :

```
call MPI_Comm_remote_size(comm,size,error)
```

Pour récupérer le groupe du communicateur distant :

```
call MPI_Comm_remote_group(comm,group,error);
```

Trainings - MPI - Organisation des processus

Rendez-vous dans le projet habituel pour les exercices , partie “organisation des processus”.

1 Introduction

2 Premiers pas avec MPI

3 Communications

4 Types dérivés

5 Groupes, communicateurs et topologies

6 Autres fonctionnalités

Les fonctionnalités que nous n'abordons pas en détail

- Copies de mémoire à mémoire
- Entrées-Sorties parallèles : confère cours dédié
- Gestion dynamique des processus
- Graphe de processus

Copies de mémoire à mémoire ou RMA

Définition :

Avoir directement accès (lecture ou écriture) à la mémoire d'un processus distant.

Mise en œuvre

1. MPI_Win_create: Définition sur chaque processus d'une zone mémoire (fenêtre mémoire locale) visible et suceptible d'être accédée par des processus distants;
2. MPI_Get/Put/Accumulate: Déclenchement du transfert des données directement de la mémoire d'un processus à celle d'un autre processus;
3. MPI_Win_fence ou MPI_Win_lock/unlock : Achèvement des transferts en cours par une étape de synchronisation, les données étant alors réellement disponibles pour les calculs;
4. MPI_Win_free : Libération de la fenêtre mémoire locale.

Entrée-Sorties parallèles: MPI-IO

Définition :

MPI-IO : interface pour gérer des opérations collectives et non bloquantes sur les fichiers

- parallélise les IOs, évite le goulet d'étranglement de leur serialization
- technique explicite pour accès non bloquants au filesystem
- opération spécifique pris en charge par l'OS : regroupement des requêtes par ex pour éviter trop nombreux petits accès discontinus

Le cours "format de données, visualisation et entrée-sorties parallèles" (11-12-13 mai 2022) propose une application avec l'utilisation de la bibliothèque HDF5 parallèle (MPI-IO en boîte noire).

Gestion dynamique des processus

Définition :

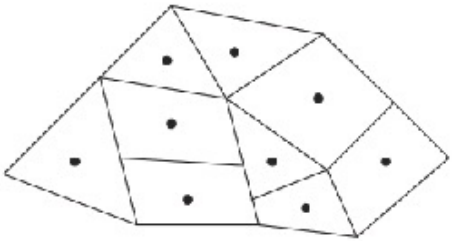
Création de processus durant l'exécution de l'application

Deux modes d'activation

- Mode maître-ouvrier : un processus active un ou plusieurs nouveaux processus (MPI_Comm_spawn()).
- Mode client-serveur : un ou plusieurs processus d'une application serveur (lancée au préalable) publient un nom de port de communication et attendent la connexion de un ou plusieurs processus d'une application cliente (lancée ultérieurement).

Graphe de processus: Topologie de type graphe

- Création d'une topologie : décomposition associée à une grille régulière
- Il existe des problèmes dont la géométrie n'est pas une grille régulière mais un domaine de forme quelconque :



- MPI fournit une fonctionnalité pour s'adapter naturellement aux géométries non-structurées : MPI_Dist_graph_create
- On peut aussi connaître le nombre et la liste des voisins de chaque processus.

Trainings - MPI - Synthèse

Rendez-vous dans le projet habituel pour les exercices , partie “synthèse”.

Les grilles de calcul

Pierre-Antoine Bouttier, Albanne Lecointre

février-mars 2023

Le cours sur les grilles de calcul et CiGri